

Linux

K19G

2026-09-04

Table of contents

Introduction	122
Linux	123
Commands — scope	123
Related books	124
How to use	124
Commands page template	124
Commands — parts at a glance	125
Editors (this book)	125
Scripting → Bash (this book)	126
Conventions	126
Formats	126
 Commands	 127
 Help and Docs	 128
Intro	129
Intro	130
Commands in this part	130
Suggested starting points	130
Related parts	130
 man	 131
Overview	131
Syntax	131
Sections (common)	131
Common Options	132
Examples with Explanations	132
Open a page	132
Disambiguate sections	132
Search when you forget the name	132
Locate the source file	133
Readable width and plain text dump	133
HTML / browser (if supported)	133
Scripting existence checks	133
Pager keys (less)	133
Notes / Pitfalls	134

2026-relevant notes	134
Related Commands	134
Additional Resources	134
help (shell builtin)	135
Overview	135
Syntax	135
Common Options / discovery helpers	135
Examples with Explanations	136
Read builtin docs	136
Builtin vs external	136
List builtins (bash)	136
Disable / re-enable a builtin (rare, debugging)	136
Conditional help in interactive setups	137
Job-control and shell language topics	137
Notes / Pitfalls	137
2026-relevant notes	137
Related Commands	138
Additional Resources	138
info	139
Overview	139
Syntax	139
Common Options	139
Navigation Keys (standalone info)	139
Examples with Explanations	140
Open a manual	140
Jump to options / a node	140
Search	140
Where is the file?	141
Non-interactive extract	141
Prefer man for quick flags, info for depth	141
Understanding the Info tree	141
Notes / Pitfalls	141
2026-relevant notes	142
Related Commands	142
Additional Resources	142
apropos	143
Overview	143
Syntax	143
Common Options	143
Examples with Explanations	144
Basic keyword search	144
AND multiple terms	144
Limit by section	144
Regex and wildcards	144
Reduce noise to section-1 commands	144

Rebuild empty database	145
Pipeline into selection	145
Compare with full-text search	145
Notes / Pitfalls	145
2026-relevant notes	145
Related Commands	146
Additional Resources	146
whatis	147
Overview	147
Syntax	147
Common Options	147
Examples with Explanations	147
Quick one-liners	147
Disambiguate by section	148
Multiple names in scripts	148
Missing database	148
Combine with type / path	148
Wildcard / regex (when supported)	148
Verify pages after package install	149
Understanding Output	149
Notes / Pitfalls	149
2026-relevant notes	149
Related Commands	149
Additional Resources	150
Files and Paths	151
Intro	152
Intro	153
Commands in this part	153
Suggested starting points	155
Related parts	156
pwd	157
Overview	157
Syntax	157
Common Options	157
Key Use Cases	157
Examples with Explanations	158
Basics	158
Capture in scripts	158
Symlink illustration	158
Prompt and logging recipes	158
Resolve then act	159
Compare with realpath	159
Directory stack awareness	159

Notes / Pitfalls	159
2026-relevant notes	159
Related Commands	160
Additional Resources	160
cd	161
Overview	161
Syntax	161
Common Options / operands	161
Key Use Cases	161
Examples with Explanations	162
Basic navigation	162
Previous directory	162
Symlink behavior	162
Paths that start with a dash	162
Script: run relative to script location	162
Script: require a directory	163
pushd / popd alternative workflow	163
One-liners	163
Understanding success and failure	163
Notes / Pitfalls	163
2026-relevant notes	164
Related Commands	164
Additional Resources	164
ls	165
Overview	165
Syntax	165
Common Options	165
Examples with Explanations	166
Everyday	166
Hidden and directories	166
Sorting and time	166
Recursive (careful)	166
Classification and inodes	166
Globs and multiple paths	167
Scripts: stable output	167
Color and aliases	167
Compare modern alternatives	167
Understanding long format	167
Notes / Pitfalls	167
2026-relevant notes	168
Related Commands	168
Additional Resources	168
tree	169
Overview	169
Syntax	169

Common Options	169
Examples with Explanations	170
Shallow overview	170
Hide dependency / VCS noise	170
Hidden files and permissions	170
Sizes	170
Full paths and filtering	170
HTML report	170
JSON for tooling (when available)	171
Compare with find / eza	171
Document a project layout	171
Notes / Pitfalls	171
2026-relevant notes	171
Related Commands	171
Additional Resources	172
mkdir	173
Overview	173
Syntax	173
Common Options	173
Key Use Cases	173
Examples with Explanations	174
Basics	174
Parents	174
Permissions	174
Brace expansion scaffolds	174
Script: ensure writable data dir	174
SELinux (when relevant)	175
Parallel with install	175
Failure cases	175
Notes / Pitfalls	175
2026-relevant notes	175
Related Commands	175
Additional Resources	176
rmdir	177
Overview	177
Syntax	177
Common Options	177
Key Use Cases	177
Examples with Explanations	178
Basic removal	178
Non-empty fails	178
Remove parent chain	178
Verbose	178
Ignore non-empty (continue batch)	178
Test emptiness in scripts	179
Combine with find (empty dirs only)	179

Brace / multi targets	179
Notes / Pitfalls	179
2026-relevant notes	179
Related Commands	180
Additional Resources	180
touch	181
Overview	181
Syntax	181
Common Options	181
Key Use Cases	181
Examples with Explanations	182
Create or refresh	182
Do not create	182
atime / mtime only	182
Date strings	182
Compact -t form	182
Reference file	183
Make and empty targets	183
Scripts: ensure path exists as empty file	183
Clear mtime into the past for cleanup tests	183
Understanding timestamps	183
Notes / Pitfalls	184
2026-relevant notes	184
Related Commands	184
Additional Resources	184
cp	185
Overview	185
Syntax	185
Common Options	185
Safety	186
Examples with Explanations	186
Files	186
Directories	186
Reflink (btrfs/xfs when available)	186
Preserve attributes selectively	186
Parents and target dir	186
Symlinks	187
One-liners	187
Notes / Pitfalls	187
2026-relevant notes	187
Related Commands	187
Additional Resources	187
mv	188
Overview	188
Syntax	188

Common Options	188
Key Use Cases	188
Examples with Explanations	189
Rename and move	189
Interactive / no-clobber	189
Atomic replace pattern	189
Verbose bulk	189
Backup on conflict	189
Directory edge cases	189
Cross-device awareness	190
Swap two files (bash)	190
Notes / Pitfalls	190
2026-relevant notes	190
Related Commands	190
Additional Resources	191
rm	192
Overview	192
Syntax	192
Common Options	192
Safety	192
Examples with Explanations	193
Files	193
Interactive	193
Directory trees	193
Empty directory only	193
Dry-run then delete with find	193
Stay on one filesystem	194
Quarantine pattern	194
Deleted but space held open	194
Notes / Pitfalls	194
2026-relevant notes	194
Related Commands	195
Additional Resources	195
cat	196
Overview	196
Syntax	196
Common Options	196
Key Use Cases	196
Examples with Explanations	197
Print files	197
Concatenate	197
Heredocs	197
Here-string and stdin	197
Show non-printing / DOS endings	198
Useless use of cat — avoid	198
Binary caution	198

Multiple files with separators (manual)	198
Notes / Pitfalls	198
2026-relevant notes	198
Related Commands	199
Additional Resources	199
less	200
Overview	200
Syntax	200
Common Options	200
Essential keys	200
Examples with Explanations	201
Open files	201
Pipe command output	201
Follow a growing log	201
Chop long lines (tables, wide logs)	202
Case-insensitive search	202
Multiple files	202
Sensible defaults via environment	202
Compare with more / bat	202
Notes / Pitfalls	202
2026-relevant notes	203
Related Commands	203
Additional Resources	203
more	204
Overview	204
Syntax	204
Common Options	204
Common keys	204
Examples with Explanations	205
Basic	205
Pipelines	205
Prefer less when available	205
Start position	205
Tiny environments	206
Compare pagers	206
Notes / Pitfalls	206
2026-relevant notes	206
Related Commands	206
Additional Resources	206
chmod	207
Overview	207
Syntax	207
Common Options	207
Mode primer	207
Safety	208

Examples with Explanations	208
Octal	208
Symbolic	208
Capital X (conditional execute)	209
Recursive the safe way	209
Reference mode	209
Diagnose	209
umask interaction (creation, not chmod)	209
Notes / Pitfalls	209
2026-relevant notes	210
Related Commands	210
Additional Resources	210
chown	211
Overview	211
Syntax	211
Common Options	211
OWNER:GROUP forms	211
Examples with Explanations	212
Basic	212
Recursive web tree	212
Reference file	212
Conditional change	212
Symlinks	212
Deploy pattern	213
Audit before change	213
Notes / Pitfalls	213
2026-relevant notes	213
Related Commands	213
Additional Resources	214
stat	215
Overview	215
Syntax	215
Common Options	215
Useful format sequences (GNU file)	215
Examples with Explanations	216
Default view	216
Script-friendly fields	216
Compare identities (same file?)	216
Timestamps	216
Filesystem info	217
Batch	217
Permissions debugging	217
Terse / portable caution	217
Notes / Pitfalls	217
2026-relevant notes	217
Related Commands	218

Additional Resources	218
file	219
Overview	219
Syntax	219
Common Options	219
Examples with Explanations	220
Basic classification	220
Brief and MIME	220
Inside compressed data	220
Symlinks	220
Scripts and interpreters	220
Stdin	220
Batch unknown directory	221
Guard uploads / pipelines	221
ELF deep dive (after file)	221
Notes / Pitfalls	221
2026-relevant notes	221
Related Commands	221
Additional Resources	222
find	223
Overview	223
Syntax	223
Common Options / Primaries	223
Key Use Cases	224
Safety	224
Examples with Explanations	224
Find by name (case-insensitive)	224
Limit depth	224
Large files on one filesystem	224
Modified in the last day	225
Exclude a directory	225
Safe batch action	225
Exec with batching	225
Permission audit (setuid)	225
Empty directories	225
Understanding Output	225
Common Usage Patterns	226
Newest N files under a tree	226
Delete with interactive dry-run	226
Notes & Pitfalls	226
Related Commands	226
Additional Resources	226
fd	227
Overview	227
Syntax	227

Common Options	227
Key Use Cases	228
Examples with Explanations	228
By extension and pattern	228
Types and hidden	228
Exec	228
Ignore rules	228
One-liner recipes	229
Notes & Pitfalls	229
2026-relevant notes	229
Comparison to alternatives	229
Related Commands	229
Additional Resources	230
locate	231
Overview	231
Syntax	231
Common Options	231
Database updates	232
Examples with Explanations	232
Basic	232
Limit and count	232
Regex	232
Basename-only	232
Existence filter	233
NUL-safe handoff	233
When locate is empty	233
locate vs find	233
Notes / Pitfalls	233
2026-relevant notes	234
Related Commands	234
Additional Resources	234
ripgrep (rg)	235
Overview	235
Syntax	235
Common Options	235
Key Use Cases	236
Examples with Explanations	236
Basics	236
Types and globs	236
Context and files	236
Unrestricted search	236
One-liner recipes	237
Notes & Pitfalls	237
2026-relevant notes	237
Comparison to alternatives	237
Related Commands	237

Additional Resources	238
ln	239
Overview	239
Syntax	239
Common Options	239
Hard vs symbolic	239
Examples with Explanations	240
Symbolic links	240
Relative symlink (GNU)	240
Hard links	240
Bulk into a directory	240
Inspect	240
Replace symlink safely	241
Common failure: relative target confusion	241
Notes / Pitfalls	241
2026-relevant notes	241
Related Commands	241
Additional Resources	242
readlink	243
Overview	243
Syntax	243
Common Options (GNU)	243
Key Use Cases	243
Examples with Explanations	244
Immediate target	244
Canonical forms	244
Multiple links	244
Script directory pattern	244
BusyBox / portability note	244
Inspect vs ls	245
NUL-safe with find	245
Notes / Pitfalls	245
2026-relevant notes	245
Related Commands	245
Additional Resources	245
realpath	246
Overview	246
Syntax	246
Common Options	246
Key Use Cases	246
Examples with Explanations	247
Absolute paths	247
Relative formatting	247
Script root	247
Same-file check	247

No symlink expand	248
Batch NUL-safe	248
Combine with find	248
Notes / Pitfalls	248
2026-relevant notes	248
Related Commands	248
Additional Resources	249
basename	250
Overview	250
Syntax	250
Common Options	250
Key Use Cases	250
Examples with Explanations	251
Basics	251
Multiple names	251
Script name	251
Strip extension patterns	251
Loop rename	251
Combine with dirname	252
NUL-safe pipeline	252
Parameter expansion alternative	252
Notes / Pitfalls	252
2026-relevant notes	252
Related Commands	253
Additional Resources	253
dirname	254
Overview	254
Syntax	254
Common Options	254
Key Use Cases	254
Examples with Explanations	255
Basics	255
Multiple arguments	255
Script directory pattern	255
Ensure parent exists	255
Sibling paths	255
Combine with realpath	256
Parameter expansion alternative (bash)	256
NUL-safe	256
Notes / Pitfalls	256
2026-relevant notes	256
Related Commands	256
Additional Resources	257
which	258
Overview	258

Syntax	258
Common Options (GNU / common)	258
Key Use Cases	258
Examples with Explanations	259
Basic	259
All matches	259
Prefer command -v in scripts	259
Compare resolution tools	259
Debug PATH order	260
Silent existence (when -s exists)	260
Notes / Pitfalls	260
2026-relevant notes	260
Related Commands	260
Additional Resources	261
alias / unalias	262
Overview	262
Syntax	262
Common Options / related	262
Examples with Explanations	263
Everyday shortcuts	263
List and inspect	263
Persist (bash)	263
Remove	263
Bypass aliases	264
sudo and alias expansion (bash)	264
Better as functions	264
Safety alias caveats	264
Notes / Pitfalls	264
2026-relevant notes	265
Related Commands	265
Additional Resources	265
install	266
Overview	266
Syntax	266
Common Options	266
Key Use Cases	267
Examples with Explanations	267
Install a script or binary	267
Create directories	267
Config with ownership	267
Leading path creation (systemd unit)	267
Multiple sources into a directory	267
Compare before replace (GNU)	268
Makefile fragment	268
Versus cp	268
Notes / Pitfalls	268

2026-relevant notes	268
Related Commands	269
Additional Resources	269
df	270
Overview	270
Syntax	270
Common Options	270
Examples with Explanations	270
Everyday overview	270
Inodes exhausted?	271
Specific path	271
Portable scripting	271
JSON-ish via findmnt (often clearer)	271
Correlate devices	271
Root reserved space	271
Understanding output	271
Notes / Pitfalls	272
2026-relevant notes	272
Related Commands	272
Additional Resources	272
du	273
Overview	273
Syntax	273
Common Options	273
Examples with Explanations	274
Top-level breakdown	274
Summaries	274
One filesystem (skip mounts)	274
Apparent vs allocated	274
Exclude noise	274
Sort biggest	274
Scripts (bytes)	275
Find large files (pair with find)	275
Notes / Pitfalls	275
2026-relevant notes	275
Related Commands	275
Additional Resources	275
bat	276
Overview	276
Syntax	276
Common Options	276
Examples with Explanations	277
View files	277
Debian binary name	277
Plain / script-friendly	277

Ranges and highlights	277
Force language	277
Multiple files and stdin	277
Git-aware	278
Themes and style	278
As man / pager helper	278
Compare with cat / less	278
Notes / Pitfalls	278
2026-relevant notes	278
Related Commands	279
Additional Resources	279
eza	280
Overview	280
Syntax	280
Common Options	280
Examples with Explanations	281
Everyday aliases	281
Sorting	281
Tree view	281
Git-aware listing	281
Headers and metadata	281
Compare with ls / tree	282
Scripts: prefer classic ls	282
Notes / Pitfalls	282
2026-relevant notes	282
Related Commands	282
Additional Resources	283
sha256sum	284
Overview	284
Syntax	284
Common Options	284
Examples with Explanations	285
Hash a file	285
Hash several files	285
Stdin	285
Write and verify a checksum file	285
Quiet / script-friendly check	285
Verify published ISO pattern	286
Directory of artifacts	286
Compare two local copies without a list	286
Notes	286
Related Commands	286
Additional Resources	287
base64	288
Overview	288

Syntax	288
Common Options	288
Examples with Explanations	288
Encode a file	288
Decode to a file	289
Encode/decode stdin pipelines	289
Small binary in a shell variable (careful)	289
Kubernetes-style secret (illustrative)	289
PEM / cert peek	289
Ignore whitespace garbage when decoding	289
Notes	290
Related Commands	290
Additional Resources	290
cmp	291
Overview	291
Syntax	291
Common Options	291
Examples with Explanations	291
Basic equal / not equal	291
Where they first differ	292
Quiet for scripts	292
List every difference (small files)	292
Compare after skipping headers	292
Limit length	292
Against a device (careful read-only)	293
Notes	293
Related Commands	293
Additional Resources	293
getfacl / setfacl	294
Overview	294
Syntax	294
Common Options	294
getfacl	294
setfacl	295
Safety	295
Examples with Explanations	295
View ACLs	295
Grant a user access	296
Grant a group and adjust mask	296
Default ACL on a directory (inherit for new files)	296
Recursive grant	296
Remove entries / clear ACLs	296
Copy ACL from one file to another	297
Dry-run	297
Backup and restore ACL tree	297
Notes	297

Related Commands	297
Additional Resources	298
chattr / lsattr	299
Overview	299
Syntax	299
Common Options	299
lsattr	299
chattr	299
Important attribute letters	300
Safety	300
Examples with Explanations	300
List attributes	300
Make a file immutable	301
Append-only log	301
Clear / set exactly	301
Protect a directory entry (directory itself)	301
No-dump flag	301
Combine with permissions	302
Notes	302
Related Commands	302
Additional Resources	302
xxd	303
Overview	303
Syntax	303
Common Options	303
Examples with Explanations	304
Basic dump	304
Limit and offset	304
Plain hex (scripts / compare)	304
C array for embedding	304
Reverse plain hex to binary	304
Edit a few bytes (workflow)	305
Inspect file magic	305
Grouping and columns for readability	305
Bits view	305
Notes	305
Related Commands	306
Additional Resources	306
strings	307
Overview	307
Syntax	307
Common Options	307
Examples with Explanations	308
Basic scan	308
Longer strings only (less noise)	308

Offsets for later xxd	308
UTF-16 little-endian (Windows-ish binaries)	308
Multiple files	308
Full file scan	308
Pipe from another command	309
Hunt secrets carefully	309
Notes	309
Related Commands	309
Additional Resources	309
namei	310
Overview	310
Syntax	310
Common Options	310
Examples with Explanations	310
Diagnose a path	310
Follow ownership along the way	311
Mount boundaries	311
Notes & Pitfalls	311
Related Commands	311
Additional Resources	311
umask	312
Overview	312
Syntax	312
Common Options	312
Key Use Cases	312
Examples with Explanations	313
Show current mask	313
How mask applies	313
Stricter group isolation	313
Secrets in a script	313
Persist for login shells	313
Notes & Pitfalls	314
Related Commands	314
Additional Resources	314
Archives and Compression	315
Intro	316
Intro	317
Commands in this part	317
Suggested starting points	317
Related parts	318
tar	319
Overview	319

Syntax	319
Common Options	319
Key Use Cases	319
Safety	320
Examples with Explanations	320
Create gzip archive	320
Create zstd archive	320
List	320
Extract	320
Strip top-level directory	320
Exclude patterns	321
Stream over ssh	321
Append is special	321
Common Usage Patterns	321
Quick directory move	321
Notes & Pitfalls	321
Related Commands	321
Additional Resources	322
gzip	323
Overview	323
Syntax	323
Common Options	323
Examples with Explanations	324
Compress / decompress	324
stdout pipelines	324
Level tradeoffs	324
Test and list	324
Recursive logs	324
View without extracting	324
Integrity in scripts	325
Notes / Pitfalls	325
2026-relevant notes	325
Related Commands	325
Additional Resources	325
xz	326
Overview	326
Syntax	326
Common Options	326
Examples with Explanations	327
Basic	327
Pipelines with tar	327
Presets and threads	327
Test and list	327
Keep and stdout	327
Memory safety on small hosts	327
Convert from gzip (recompress)	328

Notes / Pitfalls	328
2026-relevant notes	328
Related Commands	328
Additional Resources	328
bzip2	329
Overview	329
Syntax	329
Common Options	329
Examples with Explanations	330
Compress / decompress	330
Pipelines	330
Levels	330
Test	330
Search compressed	330
Prefer modern tools when choosing	330
Notes / Pitfalls	331
2026-relevant notes	331
Related Commands	331
Additional Resources	331
zstd	332
Overview	332
Syntax	332
Common Options	332
Examples with Explanations	333
Basic	333
Levels and threads	333
Pipelines with tar	333
Integrity	333
Remove source when desired	333
Dictionaries (small similar files)	334
Notes / Pitfalls	334
2026-relevant notes	334
Related Commands	334
Additional Resources	334
zip	335
Overview	335
Syntax	335
Common Options	335
Examples with Explanations	336
Create archives	336
Update and list	336
Flatten paths	336
Store without compression	336
Encryption (know the limits)	336
Split archives	336

Exclude junk	336
Date-stamped backup	337
Notes / Pitfalls	337
2026-relevant notes	337
Related Commands	337
Additional Resources	337
unzip	338
Overview	338
Syntax	338
Common Options	338
Examples with Explanations	339
List and test	339
Extract	339
Selective extract	339
Overwrite policies	339
Flatten paths	339
Pipe / quiet automation	339
Untrusted archives — path safety	340
Password	340
Notes / Pitfalls	340
2026-relevant notes	340
Related Commands	340
Additional Resources	341
zcat	342
Overview	342
Syntax	342
Examples with Explanations	342
Read a gzip log	342
Concatenate multiple	342
Portable explicit form	342
Notes & Pitfalls	343
Related Commands	343
Additional Resources	343
zgrep	344
Overview	344
Syntax	344
Examples with Explanations	344
Search compressed logs	344
Recursive pattern over mixed trees	344
Count matches	344
Notes & Pitfalls	345
Related Commands	345
Additional Resources	345

System Information	346
Intro	347
Intro	348
Commands in this part	348
Suggested starting points	349
Related parts	349
uname	351
Overview	351
Syntax	351
Common Options	351
Examples with Explanations	351
Everyday	351
Architecture branching	352
Kernel version checks	352
vs os-release	352
Container note	352
Compact inventory line	352
Notes / Pitfalls	352
2026-relevant notes	353
Related Commands	353
Additional Resources	353
hostname	354
Overview	354
Syntax	354
Common Options	354
Examples with Explanations	355
Read	355
Prefer hostnamectl on systemd	355
Temporary classic set (legacy)	355
Scripts	355
Correlate with DNS / addresses	355
Notes / Pitfalls	356
2026-relevant notes	356
Related Commands	356
Additional Resources	356
hostnamectl	357
Overview	357
Syntax	357
Common Commands / Options	357
Examples with Explanations	358
Status	358
Set hostname	358
Pretty name	358
Transient (temporary)	358

Chassis / location metadata	358
Remote / container	358
Verify files	359
Notes / Pitfalls	359
2026-relevant notes	359
Related Commands	359
Additional Resources	359
lsb_release	360
Overview	360
Syntax	360
Common Options	360
Examples with Explanations	360
Everyday	360
Short for scripts	361
Prefer os-release (portable modern)	361
Compare sources	361
Install if missing	361
Notes / Pitfalls	361
2026-relevant notes	362
Related Commands	362
Additional Resources	362
uptime	363
Overview	363
Syntax	363
Common Options	363
Examples with Explanations	363
Default	363
Pretty / since	364
Interpret load vs CPUs	364
Watch over time	364
Scripts	364
Correlate with other tools	364
Containers	364
Understanding load average	364
Notes / Pitfalls	365
2026-relevant notes	365
Related Commands	365
Additional Resources	365
date	366
Overview	366
Syntax	366
Common Options	366
Useful format sequences	366
Examples with Explanations	367
Now	367

Convert / relative	367
Filenames and logs	367
File mtime	367
Timezone one-off	367
Set clock (prefer NTP)	368
GNU vs scripts	368
Notes / Pitfalls	368
2026-relevant notes	368
Related Commands	368
Additional Resources	368
timedatectl	369
Overview	369
Syntax	369
Common Commands / Options	369
Key Use Cases	369
Examples with Explanations	370
Status	370
Machine-readable	370
List and set timezone	370
Enable NTP (systemd-timesyncd path)	370
Chrony hosts	370
Manual time (maintenance / air-gap)	371
RTC in UTC (recommended on Linux)	371
Verify sync health	371
date interaction	371
Understanding Output	372
Notes & Pitfalls	372
Related Commands	372
Additional Resources	372
cal	373
Overview	373
Syntax	373
Common Options (util-linux / common)	373
Examples with Explanations	374
Current month	374
Specific month / year	374
Week numbers for planning	374
Surrounding months	374
Historical curiosity	374
Scripts: parse carefully	374
ncal alternative	375
Notes / Pitfalls	375
2026-relevant notes	375
Related Commands	375
Additional Resources	375

whoami	376
Overview	376
Syntax	376
Common Options	376
Key Use Cases	376
Examples with Explanations	377
Basics	377
Script guards	377
Contrast with id / logname	377
Logging	377
Containers	377
Notes / Pitfalls	378
2026-relevant notes	378
Related Commands	378
Additional Resources	378
 id	 379
Overview	379
Syntax	379
Common Options	379
Examples with Explanations	379
Default	379
Specific fields	380
Other users	380
Root check in scripts	380
Real vs effective (setuid context)	380
SELinux	380
Format for logs	380
Permission debugging pattern	381
Notes / Pitfalls	381
2026-relevant notes	381
Related Commands	381
Additional Resources	381
 who	 382
Overview	382
Syntax	382
Common Options	382
Examples with Explanations	383
Who is on the system	383
Boot time	383
Yourself	383
Compare with w	383
Historical	383
Scripting inventory	383
Notes / Pitfalls	384
2026-relevant notes	384
Related Commands	384

Additional Resources	384
w	385
Overview	385
Syntax	385
Common Options	385
Examples with Explanations	385
Default	385
Filter one user	386
Short / no header	386
IP addresses	386
Correlate with load tools	386
When you need history	386
Understanding columns	386
Notes / Pitfalls	387
2026-relevant notes	387
Related Commands	387
Additional Resources	387
last	388
Overview	388
Syntax	388
Common Options	388
Examples with Explanations	388
Recent logins	388
One user	389
Reboots and shutdowns	389
Bad logins	389
Rotate awareness	389
SSH audit starter	389
Notes / Pitfalls	390
2026-relevant notes	390
Related Commands	390
Additional Resources	390
env	391
Overview	391
Syntax	391
Common Options	391
Examples with Explanations	391
Print environment	391
Run with extra variables	392
Clear environment	392
Unset for one command	392
Shebang portability	392
Chdir then run (when supported)	392
NUL-safe dump	392
Notes / Pitfalls	393

2026-relevant notes	393
Related Commands	393
Additional Resources	393
free	394
Overview	394
Syntax	394
Common Options	394
Examples with Explanations	394
Everyday	394
Watch live	395
Scripts	395
Interpret with pressure tools	395
Containers / cgroups	395
Understanding the columns	395
Notes / Pitfalls	396
2026-relevant notes	396
Related Commands	396
Additional Resources	396
lscpu	397
Overview	397
Syntax	397
Common Options	397
Examples with Explanations	397
Overview	397
Parseable / JSON	398
Core counts for scripts	398
Virtualization and flags	398
NUMA	398
Security mitigations summary	398
Containers	398
Notes / Pitfalls	399
2026-relevant notes	399
Related Commands	399
Additional Resources	399
lsmem	400
Overview	400
Syntax	400
Common Options	400
Output Columns	400
Key Use Cases	401
Examples with Explanations	401
Example 1: Basic Memory Information	401
Example 2: Human-Readable Sizes	401
Example 3: Summary Only	401
Understanding Memory States	401

Memory Block Management	401
Common Usage Patterns	402
NUMA Memory Information	402
Memory Hotplug Operations	402
Performance Analysis	402
Related Commands	402
Additional Resources	403
Best Practices	403
Scripting Examples	403
Memory Zones	403
System Integration	403
Troubleshooting	404
Advanced Usage	404
Memory Block Operations	404
Integration Examples	405
Output Formatting	405
lsusb	406
Overview	406
Syntax	406
Common Options	406
Examples with Explanations	406
List devices	406
Filter	407
Verbose probe	407
Correlate with kernel messages	407
Find device nodes	407
Tree + drivers	407
Notes / Pitfalls	407
2026-relevant notes	408
Related Commands	408
Additional Resources	408
lsmod	409
Overview	409
Syntax	409
Examples with Explanations	409
List modules	409
Search for a driver	409
Dependency / use count	409
Correlate with hardware	410
Module details	410
Unload safety check	410
Blacklist context	410
Understanding columns	410
Notes / Pitfalls	411
2026-relevant notes	411
Related Commands	411

Additional Resources	411
hwinfo	412
Overview	412
Syntax	412
Common Options	412
Key Use Cases	412
Examples with Explanations	413
Example 1: Brief Summary	413
Example 2: CPU Info	413
Example 3: Storage Info	413
Understanding Output	413
Common Usage Patterns	413
Performance Analysis	414
Related Commands	414
Additional Resources	414
Hardware Categories	414
Best Practices	414
 Processes and Jobs	 415
Intro	416
Intro	417
Commands in this part	417
Suggested starting points	418
Related parts	419
 ps	 420
Overview	420
Syntax	420
Common Invocations	420
Key Use Cases	420
Examples with Explanations	421
Classic full lists	421
Top memory and CPU	421
Custom columns	421
By user / name	421
Tree view	421
One PID in detail	422
Threads	422
Full command line (no truncation)	422
Zombies and D-state	422
Script: fail if process missing	422
Understanding Output	422
Notes & Pitfalls	423
Related Commands	423
Additional Resources	423

pstree	424
Overview	424
Syntax	424
Common Options	424
Examples with Explanations	425
Basics	425
One user or PID	425
Show ancestry of a process	425
Avoid compaction	425
Threads vs processes	425
SELinux	425
Compare with ps	426
Notes / Pitfalls	426
2026-relevant notes	426
Related Commands	426
Additional Resources	426
 top	 427
Overview	427
Syntax	427
Common Options	427
Interactive keys (procp-ng)	427
Key Use Cases	428
Examples with Explanations	428
Example: interactive default	428
Example: refresh every two seconds	428
Example: single-user focus	428
Example: watch specific PIDs	429
Example: batch sample for logs / tickets	429
Example: sort by memory from the CLI	429
Example: threads view	429
Example: kill from inside top	429
Example: one-shot CPU leaders without staying in top	429
Understanding Output	430
Notes & Pitfalls	430
Related Commands	430
Additional Resources	431
 htop	 432
Overview	432
Syntax	432
Common Options	432
Useful keys	432
Key Use Cases	433
Examples with Explanations	433
Example: start htop	433
Example: only your processes	433
Example: tree view for process storms	434

Example: attach to known PIDs	434
Example: slower refresh on overloaded boxes	434
Example: kill workflow inside htop	434
Example: nice a batch job interactively	434
Example: setup persistence	434
Understanding Output	435
Notes & Pitfalls	435
Related Commands	435
Additional Resources	435
procs	436
Overview	436
Syntax	436
Common Options	436
Examples with Explanations	437
Everyday	437
Sorting	437
Columns	437
JSON for tooling	437
Compare with ps	437
Watch like top (lightweight)	437
Notes / Pitfalls	438
2026-relevant notes	438
Related Commands	438
Additional Resources	438
pgrep	439
Overview	439
Syntax	439
Common Options	439
Safety	440
Key Use Cases	440
Examples with Explanations	440
Example: list matching PIDs	440
Example: exact name match	440
Example: full command line	441
Example: filter by user	441
Example: count for monitoring	441
Example: newest / oldest only	441
Example: children of a parent	441
Example: delimiter for other tools	441
Example: pkill graceful then force	442
Example: pkill with full-line match	442
Example: avoid ps grep	442
Understanding Output	442
Notes & Pitfalls	442
Related Commands	443
Additional Resources	443

pidof	444
Overview	444
Syntax	444
Common Options	444
Examples with Explanations	444
Basics	444
Single PID	445
Scripts	445
Existence gate	445
Omit PIDs	445
Feed to kill	445
Compare tools	445
Notes / Pitfalls	446
2026-relevant notes	446
Related Commands	446
Additional Resources	446
 kill	 447
Overview	447
Syntax	447
Common Options	447
Safety	448
Key Use Cases	448
Examples with Explanations	448
Example: graceful stop (default TERM)	448
Example: TERM, wait, then KILL	448
Example: force kill only when necessary	449
Example: reload via HUP	449
Example: signal by name via pgrep	449
Example: stop / continue (debug freeze)	449
Example: list signals	450
Example: multiple PIDs	450
Example: permission check without killing	450
Understanding Output	450
Notes & Pitfalls	450
Related Commands	451
Additional Resources	451
 killall	 452
Overview	452
Syntax	452
Common Options	452
Safety	452
Key Use Cases	453
Examples with Explanations	453
Example: graceful stop by name	453
Example: TERM then KILL	453
Example: only your processes	453

Example: interactive confirm	453
Example: verbose + wait	454
Example: HUP reload pattern	454
Example: case-insensitive / regex	454
Example: list signals	454
Example: dry reconnaissance first	454
Notes & Pitfalls	455
Related Commands	455
Additional Resources	455
jobs	456
Overview	456
Syntax	456
Common Options (bash)	456
Job states	456
Examples with Explanations	457
List jobs	457
Act on a job	457
Status changes	457
wait integration	457
Enable job control in scripts (rare)	457
disown removes from table	458
Notes / Pitfalls	458
2026-relevant notes	458
Related Commands	458
Additional Resources	458
fg	459
Overview	459
Syntax	459
Job specification	459
Examples with Explanations	459
After Ctrl-Z	459
Pull background job forward	460
Switch between jobs	460
With pipelines	460
Script note	460
Notes / Pitfalls	460
2026-relevant notes	461
Related Commands	461
Additional Resources	461
bg	462
Overview	462
Syntax	462
Job specification	462
Common Options	462

Examples with Explanations	463
Suspend then background	463
Start in background directly	463
Specific job	463
Multiple jobs	463
Typical workflow	463
Disown / survive hangup	464
Not for scripts by default	464
Notes / Pitfalls	464
2026-relevant notes	464
Related Commands	464
Additional Resources	465
nohup	466
Overview	466
Syntax	466
Common Options	466
Behavior (what you actually care about)	466
Key Use Cases	467
Examples with Explanations	467
Example: basic background job	467
Example: explicit log file	467
Example: find and stop later	467
Example: disown alternative (bash)	467
Example: combine with nice / ionice	468
Example: remote one-liner over SSH	468
Example: check nohup.out	468
Safety	468
Notes & Pitfalls	468
Related Commands	469
Additional Resources	469
timeout	470
Overview	470
Syntax	470
Duration format	470
Common Options	470
Exit status (GNU coreutils)	471
Safety	471
Key Use Cases	471
Examples with Explanations	471
Example: basic time limit	471
Example: TERM then KILL grace period	472
Example: choose the first signal	472
Example: capture timeout in scripts	472
Example: verbose diagnosis	472
Example: network tool guardrails	472
Example: command that starts with a dash	473

Example: foreground mode	473
Notes & Pitfalls	473
Related Commands	473
Additional Resources	473
watch	474
Overview	474
Syntax	474
Common Options	474
Key Use Cases	474
Examples with Explanations	475
Example: basic interval monitor	475
Example: highlight changes	475
Example: multi-command pipeline	475
Example: systemd / journal peek	475
Example: exit when output changes	475
Example: stop on error	476
Example: colors from modern CLIs	476
Example: precise interval	476
Example: exec mode (no shell)	476
Notes & Pitfalls	476
Related Commands	477
Additional Resources	477
sleep	478
Overview	478
Syntax	478
Time suffixes (GNU)	478
Examples with Explanations	478
Basics	478
Simple retry loop	479
Exponential backoff (bash)	479
Rate limiting	479
Background heartbeat	479
Interruptible	479
Portable note	480
Notes / Pitfalls	480
2026-relevant notes	480
Related Commands	480
Additional Resources	480
nice	481
Overview	481
Syntax	481
Common Options	481
Safety	482
Key Use Cases	482

Examples with Explanations	482
Example: start a polite batch job	482
Example: background + log	482
Example: renice a running PID	482
Example: renice all of a user's jobs	483
Example: observe nice in ps / top	483
Example: nice + ionice + nohup combo	483
Example: systemd equivalent knobs	483
Example: print current nice (when supported)	483
Understanding Output	483
Notes & Pitfalls	484
Related Commands	484
Additional Resources	484
renice	485
Overview	485
Syntax	485
Common Options	485
Nice Values	485
Key Use Cases	486
Examples with Explanations	486
Example 1: Basic Usage	486
Example 2: User Processes	486
Example 3: Process Group	486
Understanding Output	486
Common Usage Patterns	486
Performance Analysis	487
Related Commands	487
Additional Resources	487
Best Practices	487
Use Cases	488
ionice	489
Overview	489
Syntax	489
I/O classes	489
Common Options	489
Safety	490
Key Use Cases	490
Examples with Explanations	490
Example: run command as idle I/O	490
Example: best-effort low priority	490
Example: change a running process	491
Example: query current setting	491
Example: nice + ionice together	491
Example: nohup overnight job	491
Example: try without failing scripts	491
Example: avoid realtime unless justified	492

Notes & Pitfalls	492
Related Commands	492
Additional Resources	492
fuser	493
Overview	493
Syntax	493
Common Options	493
Key Use Cases	493
Examples with Explanations	494
Mount busy	494
Ports	494
Files	494
One-liner recipes	494
Notes & Pitfalls	494
2026-relevant notes	495
Related Commands	495
Additional Resources	495
ulimit	496
Overview	496
Syntax	496
Common Options	496
Key Use Cases	497
Examples with Explanations	497
Show limits	497
Raise open files (session)	497
Cores and stack	497
One-liner recipes	497
Persistent configuration (context)	498
Notes & Pitfalls	498
2026-relevant notes	498
Related Commands	498
Additional Resources	498
ipcs	499
Overview	499
Syntax	499
Common Options	499
Examples with Explanations	500
Overview	500
Limits and usage	500
Creator and PIDs	500
Find leftovers by user	500
Cleanup companion	500
Kernel params (context)	501
Notes / Pitfalls	501
2026-relevant notes	501

Related Commands	501
Additional Resources	501
ipcrm	502
Overview	502
Syntax	502
Common Options	502
Examples with Explanations	503
Inspect then remove one segment	503
Remove by key	503
Message queues / semaphores	503
Scripted cleanup of your own leftovers	503
After crash lab example	503
Notes / Pitfalls	504
2026-relevant notes	504
Related Commands	504
Additional Resources	504
System Monitoring	505
Intro	506
Intro	507
Commands in this part	507
Suggested starting points	508
Related parts	508
vmstat	509
Overview	509
Syntax	509
Common Options	509
Key Use Cases	509
Examples with Explanations	510
Live sample (default triage)	510
Timestamped capture for logs	510
Memory summary counters	510
Disk stats mode	510
Active / inactive memory view	510
Correlate with siblings	511
Understanding Output	511
Notes & Pitfalls	512
Related Commands	512
Additional Resources	512
iostat	513
Overview	513
Syntax	513
Common Options	513

Key Use Cases	513
Examples with Explanations	514
Extended live samples (default incident command)	514
Skip since-boot line explicitly	514
One device (NVMe / disk)	514
CPU only vs devices only	514
Partitions and mapper names	514
Timestamped capture	514
Pair with process-level I/O	515
Understanding Output	515
Notes & Pitfalls	515
Related Commands	516
Additional Resources	516
mpstat	517
Overview	517
Syntax	517
Common Options	517
Key Use Cases	517
Examples with Explanations	518
Average CPU samples	518
Per-CPU breakdown (primary triage)	518
Specific cores	518
Interrupts	518
JSON for tooling	518
Pair with process CPU	519
Understanding Output	519
Notes & Pitfalls	519
Related Commands	520
Additional Resources	520
sar	521
Overview	521
Syntax	521
Common Options	521
Key Use Cases	521
Examples with Explanations	522
Live CPU samples	522
Memory and load	522
Disk and I/O rates	522
Network devices	522
Per-CPU history of “now”	522
Today’s history (collection must be enabled)	522
Slice a time window	523
Read another day’s file	523
Enable collection on Ubuntu (sketch)	523
Incident bundle	523
Understanding Output	523

Notes & Pitfalls	524
Related Commands	524
Additional Resources	524
lsof	525
Overview	525
Syntax	525
Common Options	525
Key Use Cases	525
Examples with Explanations	526
Who listens on port 22?	526
Open files for a PID	526
Deleted files still open (disk full mystery)	526
Files under a mount (before umount)	526
Network connections for a user	527
Command name filter	527
PIDs only for scripting	527
UNIX sockets / pipes (advanced)	527
Understanding Output	527
Notes & Pitfalls	528
Related Commands	528
Additional Resources	528
strace	529
Overview	529
Syntax	529
Common Options	529
Safety	530
Key Use Cases	530
Examples with Explanations	530
Why did it fail? (file open)	530
Network path of a client	530
Summary profile	530
Attach to a running PID	531
Write a full log	531
Longer strings and FD paths	531
Child processes / build systems	531
Hang diagnosis sketch	531
Understanding Output	531
Notes & Pitfalls	532
Related Commands	532
Additional Resources	532
perf	533
Overview	533
Syntax	533
Common Subcommands	533
Common Options (record / stat)	533

Key Use Cases	534
Examples with Explanations	534
High-level counters for one command	534
Live profiling	534
Record + report workflow	534
System-wide snapshot	535
One-liner recipes	535
Paranoid settings (understand before changing)	535
Notes & Pitfalls	535
2026-relevant notes	535
Comparison to alternatives	536
Related Commands	536
Additional Resources	536
btop	537
Overview	537
Syntax	537
Common Options	537
Interactive Keybindings (typical)	537
Key Use Cases	538
Examples with Explanations	538
Start defaults	538
Low-color SSH / serial console	538
Faster or slower refresh	538
One-liner recipes	539
Notes & Pitfalls	539
2026-relevant notes	539
Comparison to alternatives	539
Related Commands	539
Additional Resources	540
ltrace	541
Overview	541
Syntax	541
Common Options	541
Safety	541
Examples with Explanations	542
Trace a command	542
Summary counts	542
Attach	542
Notes & Pitfalls	542
Related Commands	542
Additional Resources	542
Users and Groups	543
Intro	544

Intro	545
Commands in this part	545
Suggested starting points	546
Related parts	546
sudo	547
Overview	547
Syntax	547
Common Options	547
Key Use Cases	548
Safety	548
Examples with Explanations	548
Run one admin command	548
Edit a system file safely	548
Root login shell when you need many steps	549
Run as another service user	549
See your privileges	549
Pass env vars explicitly (preferred over -E)	549
Credential cache control	549
Non-interactive / scripts	549
Dangerous patterns to avoid	550
Read logs of sudo use	550
Understanding Output	550
Notes & Pitfalls	550
Related Commands	550
Additional Resources	551
visudo	552
Overview	552
Syntax	552
Common Options	552
Examples with Explanations	552
Edit main sudoers	552
Edit a drop-in (preferred)	553
Check syntax	553
Common policy snippets (examples)	553
Recover from broken sudoers (prevention)	553
Test as user	553
Notes / Pitfalls	553
2026-relevant notes	554
Related Commands	554
Additional Resources	554
useradd	555
Overview	555
Syntax	555
Common Options	555
Defaults and files	556

Safety	556
Key Use Cases	556
Examples with Explanations	556
Example: human user with home and bash	556
Example: Ubuntu interactive alternative	557
Example: supplementary groups at creation	557
Example: service / system account	557
Example: fixed UID/GID for NFS	557
Example: account expiry	557
Example: custom home location	557
Example: do not create user-private group	558
Example: verify skeleton contents	558
Notes & Pitfalls	558
Related Commands	558
Additional Resources	558
usermod	559
Overview	559
Syntax	559
Common Options	559
Safety	560
Key Use Cases	560
Examples with Explanations	560
Example: append supplementary groups (-aG)	560
Example: wrong vs right group edit	560
Example: change shell	561
Example: lock and unlock	561
Example: move home directory	561
Example: rename login	561
Example: change primary group	561
Example: change UID carefully	562
Example: expiry	562
Example: service account hardening	562
Notes & Pitfalls	562
Related Commands	562
Additional Resources	563
userdel	564
Overview	564
Syntax	564
Common Options	564
Examples with Explanations	564
Delete account, keep home	564
Delete account and home	565
Refuse when logged in	565
Processes still running	565
Orphan files outside home	565
System users	565

Notes / Pitfalls	565
2026-relevant notes	566
Related Commands	566
Additional Resources	566
passwd	567
Overview	567
Syntax	567
Common Options	567
Examples with Explanations	567
Change passwords	567
Status	568
Lock / unlock	568
Force change next login	568
Scripted set (controlled environments)	568
Service accounts	568
Notes / Pitfalls	569
2026-relevant notes	569
Related Commands	569
Additional Resources	569
chage	570
Overview	570
Syntax	570
Common Options	570
Examples with Explanations	570
View policy	570
Force change on next login	571
Set max age 90 days, warn 14	571
Disable aging	571
Account expiry date	571
Batch service accounts	571
Combine with passwd	571
Notes / Pitfalls	571
2026-relevant notes	572
Related Commands	572
Additional Resources	572
groupadd	573
Overview	573
Syntax	573
Common Options	573
Related membership tools	573
Safety	574
Key Use Cases	574
Examples with Explanations	574
Example: simple project group	574
Example: fixed GID for NFS fleet	574

Example: system group for a daemon	574
Example: create group then add members (-aG)	575
Example: Ubuntu interactive alternative	575
Example: idempotent-ish create in scripts	575
Example: shared directory ownership pattern	575
Example: list and inspect	575
Notes & Pitfalls	576
Related Commands	576
Additional Resources	576
groupmod	577
Overview	577
Syntax	577
Common Options	577
Examples with Explanations	577
Rename a group	577
Change GID	578
Migrate file ownership after GID change	578
Non-unique GID (rare)	578
Verify	578
Notes / Pitfalls	578
2026-relevant notes	579
Related Commands	579
Additional Resources	579
groupdel	580
Overview	580
Syntax	580
Common Options	580
Examples with Explanations	580
Delete a group	580
Refuse if primary group	581
Check membership first	581
Automation cleanup	581
LDAP / remote groups	581
Notes / Pitfalls	581
2026-relevant notes	582
Related Commands	582
Additional Resources	582
getent	583
Overview	583
Syntax	583
Common Databases	583
Common Options	583
Key Use Cases	584
Examples with Explanations	584
Users and groups	584

Hosts	584
Services	584
One-liner recipes	584
Notes & Pitfalls	585
2026-relevant notes	585
Related Commands	585
Additional Resources	585
su	586
Overview	586
Syntax	586
Common Options	586
Safety	586
Examples with Explanations	587
Become root (login)	587
Run one command as root	587
Switch to a service user	587
Preserve selected environment	587
Notes & Pitfalls	587
Related Commands	587
Additional Resources	588
Networking	589
Intro	590
Intro	591
Commands in this part	591
Suggested starting points	593
Related parts	593
ip	594
Overview	594
Syntax	594
Common Options	594
Key Use Cases	594
Safety	595
Examples with Explanations	595
Brief overview	595
Full address list	595
Bring link up/down	595
Add/remove an address	595
Default route	596
Static route	596
Neighbor table (ARP)	596
Statistics	596
Network namespace (quick)	596
Understanding Output	596

Common Usage Patterns	597
Live IP on a server you did not configure	597
Flush all IPv4 addresses on a NIC (careful)	597
Notes & Pitfalls	597
Related Commands	597
Additional Resources	597
ss	598
Overview	598
Syntax	598
Common Options	598
Key Use Cases	598
Examples with Explanations	599
Listening TCP with processes	599
Established connections	599
Filter by port	599
Filter by address	599
UDP listeners	599
Summary and states	600
UNIX domain sockets	600
“What owns this port?”	600
IPv6 only	600
Understanding Output	600
Notes & Pitfalls	600
Related Commands	601
Additional Resources	601
ping	602
Overview	602
Syntax	602
Common Options	602
Safety	602
Examples with Explanations	603
Basic connectivity (4 probes)	603
Numeric only (faster, no rDNS noise)	603
IPv6	603
Quiet summary for scripts	603
Deadline: fail fast in automation	603
Larger payload (rough MTU / fragmentation check)	604
Source from a specific interface	604
Faster sampling (still polite)	604
Compare loss to a hop vs destination	604
Understanding Output	604
Notes & Pitfalls	605
Related Commands	605
Additional Resources	605

traceroute	606
Overview	606
Syntax	606
Common Options	606
Safety	606
Examples with Explanations	607
Basic path	607
ICMP mode (when UDP probes are blocked)	607
TCP traceroute to HTTPS (firewall-friendly)	607
Limit hop count and speed up failure	607
IPv6 path	607
Compare UDP vs TCP vs ICMP	608
Start mid-path (skip nearby hops)	608
Use a specific source address (multi-homed)	608
Understanding Output	608
Notes & Pitfalls	608
Related Commands	609
Additional Resources	609
mtr	610
Overview	610
Syntax	610
Common Options	610
Key Use Cases	611
Examples with Explanations	611
Interactive session	611
Non-interactive report (scripts & tickets)	611
TCP path to HTTPS	611
IPv6 and dual-stack	611
One-liner recipes	612
Interpreting loss (important)	612
Understanding Output	612
Notes & Pitfalls	612
2026-relevant notes	613
Comparison to alternatives	613
Related Commands	613
Additional Resources	613
dig	614
Overview	614
Syntax	614
Common Options	614
Examples with Explanations	615
Basic A record	615
Short output for scripts	615
Query a specific resolver	615
MX and TXT	615
NS and SOA (delegation / zone authority)	615

Reverse DNS (PTR)	615
Answer section only (readable)	616
Trace from the root (find where delegation breaks)	616
Follow CNAME manually	616
TCP query (firewall / large answers)	616
Non-standard port (lab DNS)	616
Batch mode	616
Check mail-related records quickly	616
Understanding Output	617
Notes & Pitfalls	617
Related Commands	617
Additional Resources	617
host	618
Overview	618
Syntax	618
Common Options	618
Key Use Cases	618
Examples with Explanations	619
Forward lookup	619
Mail and text records	619
Reverse DNS	619
Query a specific server	619
Verbose and script-friendly	619
One-liner recipes	620
Notes & Pitfalls	620
2026-relevant notes	620
Comparison to alternatives	620
Related Commands	621
Additional Resources	621
nslookup	622
Overview	622
Syntax	622
Common Options / Directives	622
Key Use Cases	622
Examples with Explanations	623
Non-interactive	623
Interactive	623
Prefer modern equivalents	623
One-liner recipes	623
Notes & Pitfalls	623
2026-relevant notes	624
Related Commands	624
Additional Resources	624
doggo	625
Overview	625

Syntax	625
Common Options (typical)	625
Examples with Explanations	626
Basic queries	626
Choose resolver	626
Reverse / other types	626
Compare with dig	626
Scripting	626
Encrypted DNS (when built-in)	627
Notes / Pitfalls	627
2026-relevant notes	627
Related Commands	627
Additional Resources	627
resolvectl	628
Overview	628
Syntax	628
Common Commands	628
Key Use Cases	628
Examples with Explanations	629
Status and query	629
Cache	629
Per-link DNS	629
One-liner recipes	629
Notes & Pitfalls	629
2026-relevant notes	630
Related Commands	630
Additional Resources	630
curl	631
Overview	631
Syntax	631
Common Options	631
Key Use Cases	632
Safety	632
Examples with Explanations	632
Download	632
API GET with pretty JSON	632
POST JSON	632
Headers only / method checks	633
Follow redirects, treat 404 as failure	633
Timing metrics	633
Auth	633
Upload file (multipart)	633
Client certificate (mTLS)	633
Proxy and retries	633
Show response headers + body separately	634
Understanding Output	634

Notes & Pitfalls	634
Related Commands	634
Additional Resources	634
wget	635
Overview	635
Syntax	635
Common Options	635
Safety	636
Examples with Explanations	636
Simple download (keeps remote name)	636
Save under a specific name	636
Resume interrupted download	636
Quiet download into a directory	636
Rate-limited fetch with retries	637
Multiple URLs from a file	637
Spider: check if URL is reachable	637
Authenticated download (prefer netrc)	637
Custom header (e.g. token) without logging token in shell history carefully	637
One-page offline copy (HTML + assets)	637
Bounded recursive mirror of a docs tree	638
Timestamped re-sync of a single artifact	638
Understanding Output	638
Notes & Pitfalls	638
Related Commands	638
Additional Resources	639
ssh	640
Overview	640
Syntax	640
Common Options	640
Key Use Cases	640
Safety	641
Examples with Explanations	641
Login	641
Remote command	641
Custom port / options	641
ProxyJump via bastion	641
Local port forward (remote DB to local)	641
Dynamic SOCKS proxy	642
Config file (~/.ssh/config)	642
Verbose auth debug	642
Common Usage Patterns	642
Copy key (once)	642
Escape sequence (client)	642
Notes & Pitfalls	642
Related Commands	643
Additional Resources	643

scp	644
Overview	644
Syntax	644
Common Options	644
Safety	644
Examples with Explanations	645
Local → remote file	645
Remote → local	645
Directory tree	645
Custom port and key	645
Preserve times/modes	645
Via jump host (bastion)	645
Bandwidth-limited copy	646
Multiple sources to a remote directory	646
Remote-to-remote through your laptop	646
Verbose debug when auth fails	646
Understanding Output	646
Notes & Pitfalls	646
Related Commands	647
Additional Resources	647
 rsync	 648
Overview	648
Syntax	648
Common Options	648
Safety	649
Examples with Explanations	649
Dry-run first (always for delete or production)	649
Local archive backup with progress	649
Push project over SSH	649
Pull logs from a server	650
Mirror with excludes	650
Throttled backup (shared link)	650
One filesystem only (skip other mounts)	650
Checksum mode when clocks or touch times lie	650
Show only what would change (itemize)	650
Understanding Output	650
Notes & Pitfalls	651
Related Commands	651
Additional Resources	651
 nc	 652
Overview	652
Syntax	652
Common Options (OpenBSD nc on Ubuntu)	652
Safety	652
Examples with Explanations	653
Test if a TCP port is open	653

HTTP GET by hand	653
TLS is not built into classic nc	653
Simple file transfer (lab / trusted LAN)	653
Listen and show inbound banner/data	654
UDP probe	654
Port range scan (light)	654
Pipe program output to a remote listener	654
Bind to a specific local address	654
Chatty timeout for scripts	654
Understanding Output	655
Notes & Pitfalls	655
Related Commands	655
Additional Resources	655
tcpdump	656
Overview	656
Syntax	656
Common Options	656
Safety	657
Filter expressions (BPF) essentials	657
Examples with Explanations	657
See anything on default route interface	657
HTTPS-related traffic for one host	657
DNS queries and responses	657
Only SYN packets (connection attempts)	658
Capture full packets to file for Wireshark	658
Read a pcap back in the terminal	658
HTTP cleartext headers (lab only)	658
ICMP (ping) traffic	658
Traffic involving a local port (e.g. reverse proxy)	658
Limit and timestamp deltas	658
Exclude SSH so your session does not flood the capture	659
Understanding Output	659
Notes & Pitfalls	659
Related Commands	659
Additional Resources	660
nmap	661
Command Overview	661
Syntax	661
Common Options	661
Key Use Cases	662
Examples with Explanations	662
1. Basic Scan	662
2. Intensive Scan	662
3. Stealth Scan	663
4. Service Version Detection	663
5. OS Detection	663

Understanding Nmap	663
Scan Types	663
Port Selection	664
Host Discovery	664
Version Detection	664
Script Scanning	665
Output Formats	665
Performance Tuning	665
Advanced Techniques	666
Firewall Evasion	666
NSE Scripts Examples	666
Best Practices	667
Related Commands	667
Additional Resources	667
openssl	668
Overview	668
Syntax	668
Common Subcommands & Options	668
s_client (frequent flags)	668
x509 (frequent flags)	669
Safety	669
Examples with Explanations	669
Version and providers	669
Probe a website's certificate	670
Full chain and leaf details	670
SHA-256 fingerprint	670
Inspect a local PEM cert	670
STARTTLS (mail example)	670
Generate lab key + self-signed cert	670
CSR for an external CA	671
PEM DER	671
Digests (alternate to *sum tools)	671
Random secrets	671
Quick HTTPS port check (banner/handshake only)	671
Notes	671
Related Commands	672
Additional Resources	672
nmcli	673
Overview	673
Syntax	673
Common Commands	673
Key Use Cases	673
Examples with Explanations	674
Overview	674
Bounce a connection	674
Static Ethernet profile	674

DHCP Ethernet	674
Wi-Fi	674
Reload and DNS insight	674
One-liner recipes	675
Notes & Pitfalls	675
2026-relevant notes	675
Related Commands	675
Additional Resources	675
ufw	676
Overview	676
Syntax	676
Common Commands	676
Key Use Cases	676
Safety	677
Examples with Explanations	677
Safe first enable	677
Web and custom ports	677
From management network only	677
App profiles	677
Numbered delete / limit	677
One-liner recipes	678
Notes & Pitfalls	678
2026-relevant notes	678
Related Commands	678
Additional Resources	678
nft	679
Overview	679
Syntax	679
Common Commands	679
Safety	680
Examples with Explanations	680
See what is active	680
Minimal inet filter sketch (lab / understanding)	680
Sets for many ports/IPs	681
Delete a rule by handle	681
Save and restore	681
Persist on Ubuntu (high level)	681
Coexistence checks	681
JSON for scripts	682
Notes	682
Related Commands	682
Additional Resources	682
iptables	683
Overview	683
Syntax	683

Tables & common chains	683
Common Options	683
Safety	684
Examples with Explanations	684
List	684
Allow SSH, HTTP; default drop (sketch)	684
NAT masquerade (router)	684
Save (Debian/Ubuntu example)	684
Prefer UFW on Ubuntu	685
Notes	685
Related Commands	685
ifconfig	686
Overview	686
Syntax	686
Common Options / forms	686
Examples with Explanations	687
Display	687
Preferred modern equivalents	687
Bring up/down	687
Set address (legacy)	687
MTU	687
Install on Debian/Ubuntu if missing	688
Notes / Pitfalls	688
2026-relevant notes	688
Related Commands	688
Additional Resources	688
netstat	689
Overview	689
Syntax	689
Common Options	689
Examples with Explanations	690
Listening services	690
Established connections	690
Routing	690
Interfaces / stats	690
Continuous	690
Translate old → new	690
Notes / Pitfalls	691
2026-relevant notes	691
Related Commands	691
Additional Resources	691
ftp	692
Overview	692
Syntax	692
Common Options / commands	692

Examples with Explanations	693
Interactive session	693
Prefer SFTP instead	693
lftp (better modern client)	693
Scripted (discouraged for secrets)	693
Firewall note	693
Notes / Pitfalls	694
2026-relevant notes	694
Related Commands	694
Additional Resources	694
telnet	695
Overview	695
Syntax	695
Common Options	695
Examples with Explanations	695
Port open check	695
Speak HTTP manually	696
SMTP banner grab (authorized testing only)	696
Escape and quit	696
Prefer TLS-aware tools for HTTPS/SMTPS	696
Notes / Pitfalls	696
2026-relevant notes	697
Related Commands	697
Additional Resources	697
ssh-keygen	698
Overview	698
Syntax	698
Common Options	698
Safety	698
Examples with Explanations	699
Create an Ed25519 key	699
RSA for legacy only	699
Fingerprint	699
Fix / inspect known_hosts	699
Public key from private	699
Related Commands	699
Additional Resources	700
ssh-copy-id	701
Overview	701
Syntax	701
Common Options	701
Safety	701
Examples with Explanations	702
Default identity	702
Explicit key and port	702

Via ProxyJump	702
Manual equivalent (when tool missing)	702
Notes & Pitfalls	702
Related Commands	702
Additional Resources	703
sftp	704
Overview	704
Syntax	704
Common Options	704
Examples with Explanations	704
Interactive session	704
Batch download	705
Jump host	705
Non-interactive one-shot via ssh	705
Notes & Pitfalls	705
Related Commands	705
Additional Resources	706
ethtool	707
Overview	707
Syntax	707
Common Options	707
Safety	707
Examples with Explanations	708
Link status	708
Driver info	708
Statistics	708
Identify physical port	708
Offloads	708
Persist settings	708
Related Commands	708
Additional Resources	709
firewall-cmd	710
Overview	710
Syntax	710
Common Options / Patterns	710
Safety	710
Examples with Explanations	711
Inventory	711
Allow HTTP permanently	711
Open a custom port	711
Rich rule example	711
Runtime vs permanent	711
Notes & Pitfalls	711
Related Commands	712
Additional Resources	712

Scheduling	713
Intro	714
Intro	715
Commands in this part	715
Suggested starting points	715
Related parts	715
cron	716
Overview	716
Syntax	716
Common schedule patterns	716
Key Use Cases	717
Safety	717
Examples with Explanations	717
User crontab (via crontab tool)	717
Example job lines	717
System-wide locations (Ubuntu)	718
Environment inside cron	718
Debugging “it didn’t run”	718
Prefer systemd timers when...	719
Understanding Output	719
Notes & Pitfalls	719
Related Commands	719
Additional Resources	720
crontab	721
Overview	721
Syntax	721
Common Options	721
Key Use Cases	721
Safety	722
Examples with Explanations	722
List and edit	722
Install from a file (idempotent in config mgmt)	722
Append a line carefully	722
Remove with confirmation	723
Root manages another user	723
Validate schedule fragments	723
Common job examples	723
Understanding Output	723
Notes & Pitfalls	724
Related Commands	724
Additional Resources	724
anacron	725
Overview	725
Syntax	725

Common Options	725
Configuration	725
Examples with Explanations	726
Status / integration	726
Force a run (careful)	726
Update timestamps only	726
Put a job in daily.d style	726
Prefer systemd timer for new work	726
Notes / Pitfalls	727
2026-relevant notes	727
Related Commands	727
Additional Resources	727
at	728
Overview	728
Syntax	728
Common Options	728
Time expressions	729
Examples with Explanations	729
Interactive job	729
One-liner from pipe	729
From file	729
Queue management	729
batch	730
Ensure service	730
Notes / Pitfalls	730
2026-relevant notes	730
Related Commands	730
Additional Resources	730
Logging	731
Intro	732
Intro	733
Commands in this part	733
Suggested starting points	733
Related parts	733
dmesg	734
Overview	734
Syntax	734
Common Options	734
Examples with Explanations	735
Recent kernel messages	735
Follow live (plug hardware)	735
Errors and warnings	735
OOM investigation	735

USB / disk events	735
Clear buffer (careful)	735
Permissions	736
Notes / Pitfalls	736
2026-relevant notes	736
Related Commands	736
Additional Resources	736
logger	737
Overview	737
Syntax	737
Common Options	737
Examples with Explanations	738
Simple messages	738
From scripts	738
Pipe multi-line output	738
Priority levels	738
View on systemd	738
stderr mirror	739
Structured-ish with journald (when supported)	739
Notes / Pitfalls	739
2026-relevant notes	739
Related Commands	739
Additional Resources	739
logrotate	740
Overview	740
Syntax	740
Common Options	740
Common config directives	740
Key Use Cases	741
Examples with Explanations	741
Dry-run entire system config	741
Force one drop-in (verbose)	741
Inspect state	741
Example application drop-in	742
copytruncate when the app won't reopen	742
size-based emergency pattern	742
When does it run? (Ubuntu)	743
Disk pressure triage with logs	743
Understanding Output	743
Notes & Pitfalls	743
Related Commands	744
Additional Resources	744

Hardware	745
Intro	746
Intro	747
Commands in this part	747
Suggested starting points	747
Related parts	747
lshw	749
Overview	749
Syntax	749
Common Options	749
Key Use Cases	749
Examples with Explanations	750
Quick overview	750
Network and disks	750
Memory and CPU	750
Export	750
One-liner recipes	750
Notes & Pitfalls	751
2026-relevant notes	751
Related Commands	751
Additional Resources	751
lspci	752
Overview	752
Syntax	752
Common Options	752
Key Use Cases	752
Examples with Explanations	753
Everyday list	753
Drivers and modules	753
Network / GPU filters	753
One device, deep detail	753
Tree view	753
Machine-readable	754
ID database refresh	754
Correlate with sysfs / ethtool	754
Understanding Output	754
Notes & Pitfalls	755
Related Commands	755
Additional Resources	755
dmidecode	756
Overview	756
Syntax	756
Common Options	756
Key Use Cases	756

Examples with Explanations	757
System identity	757
BIOS / firmware	757
Memory population	757
Processor summary	757
Baseboard and chassis	757
Inventory one-liners for scripts	758
Compare with sysfs (no full dump)	758
Understanding Output	758
Notes & Pitfalls	758
Related Commands	759
Additional Resources	759
hdparm	760
Overview	760
Syntax	760
Common Options	760
Examples with Explanations	761
Identify	761
Power status	761
Read-only info queries (safer)	761
Benchmarks (I/O heavy; maintenance window)	761
Write cache (know the risk)	761
Reread partitions	761
Notes / Pitfalls	762
2026-relevant notes	762
Related Commands	762
Additional Resources	762
 Storage and Filesystems	 763
Intro	764
 Intro	 765
Commands in this part	765
Suggested starting points	766
Related parts	767
 lsblk	 768
Overview	768
Syntax	768
Common Options	768
Key Use Cases	769
Examples with Explanations	769
Everyday inventory	769
UUID-ready view for fstab	769
Hide loop noise (snaps)	769
JSON for scripts	770

Topology / rotational / transport	770
One device subtree	770
Align with blkid and mounts	770
Empty or new disk check	770
Understanding Output	770
Notes & Pitfalls	771
Related Commands	771
Additional Resources	771
blkid	772
Overview	772
Syntax	772
Common Options	772
Key Use Cases	772
Examples with Explanations	773
List all probed devices	773
One device	773
Export form (shell-friendly)	773
UUID only (fstab snippets)	773
Lookup by UUID or LABEL	773
Compare with lsblk	774
After mkfs verification	774
udev property style	774
Understanding Output	774
Notes & Pitfalls	775
Related Commands	775
Additional Resources	775
findmnt	776
Overview	776
Syntax	776
Common Options	776
Key Use Cases	777
Examples with Explanations	777
Full tree	777
df-like sizes	777
What holds this path?	777
By source or target	777
Script-friendly	778
Busy mount debugging	778
Verify fstab	778
NFS / bind focus	778
Understanding Output	778
Notes & Pitfalls	779
Related Commands	779
Additional Resources	779

fdisk	780
Overview	780
Syntax	780
Common Options	780
Safety	781
Key Use Cases	781
Examples with Explanations	781
List all disks (read-only)	781
Confirm target before editing	781
Interactive GPT data disk (sketch)	782
Kernel re-read after write	782
Next steps after partitioning	782
Scripted dump / restore (sfdisk)	782
EFI system partition type (boot disk work)	783
Understanding Output	783
Notes & Pitfalls	783
Related Commands	783
Additional Resources	783
 mkfs	 784
Overview	784
Syntax	784
Common Options	784
Safety	785
Key Use Cases	785
Examples with Explanations	785
Verify target, then ext4 data disk	785
XFS data volume	785
FAT32 for USB interoperability	786
Reduce ext* reserved space on large data disks	786
LVM logical volume	786
Capture UUID for fstab	786
Whole-disk vs partition	786
Understanding Output	787
Notes & Pitfalls	787
Related Commands	787
Additional Resources	787
 fsck	 788
Overview	788
Syntax	788
Common Options	788
Safety	788
Key Use Cases	789
Examples with Explanations	789
Report-only on unmounted ext4	789
Interactive repair	789
Auto-yes repair (after backup / when you accept loss)	789

Dry-run which helpers would run	790
Force ext4 check with progress	790
Root filesystem	790
XFS path	790
Mount and verify after clean check	790
Understanding Output / Exit codes	791
Notes & Pitfalls	791
Related Commands	791
Additional Resources	792
mount	793
Overview	793
Syntax	793
Common Options	793
Key Use Cases	793
Safety	794
Examples with Explanations	794
Inventory	794
Mount a partition by UUID	794
Read-only mount (forensics / recovery)	794
NFS	795
CIFS / SMB	795
Bind mounts	795
tmpfs	795
Remount	795
fstab-driven	795
Overlay (container-style)	796
Understanding Output	796
Notes & Pitfalls	796
Related Commands	796
Additional Resources	797
umount	798
Overview	798
Syntax	798
Common Options	798
Key Use Cases	798
Safety	799
Examples with Explanations	799
Unmount by mountpoint (preferred)	799
Unmount by device	799
Confirm before and after	799
Busy mount diagnosis	799
Lazy unmount	800
Force NFS (last resort)	800
Remount read-only fallback	800
Multiple mounts / bind trees	800
Before fsck	800

Understanding Output	800
Notes & Pitfalls	801
Related Commands	801
Additional Resources	801
dd	802
Overview	802
Syntax	802
Common Options	802
Safety	803
Examples with Explanations	803
List devices before any write	803
Write an ISO to a USB stick (destructive to the stick)	803
Create a raw disk image (backup a USB)	804
Restore an image to a USB	804
Sparse empty file (test disk image)	804
Sample first megabyte (forensics / header peek)	804
Damaged media (best-effort copy)	805
Progress with pv (when installed)	805
Wipe a USB free-space / reuse (destructive)	805
Notes	805
Related Commands	805
Additional Resources	806
sync	807
Overview	807
Syntax	807
Common Options	807
Safety	807
Examples with Explanations	808
Flush everything (classic)	808
After writing a USB with dd	808
Sync specific file (GNU)	808
Safe removable-media workflow	808
In scripts before reboot/power events	808
Drop caches (admin debug — not for production routine)	809
Notes	809
Related Commands	809
Additional Resources	809
losetup	810
Overview	810
Syntax	810
Common Options	810
Key Use Cases	810
Examples with Explanations	811
Attach and list	811
Mount ISO	811

Partitions inside image	811
Detach	811
One-liner recipes	811
Notes & Pitfalls	812
2026-relevant notes	812
Related Commands	812
Additional Resources	812
duf	813
Overview	813
Syntax	813
Common Options	813
Examples with Explanations	814
Everyday	814
Inodes	814
JSON	814
Hide noise	814
Compare with df	814
Threshold awareness	814
Notes / Pitfalls	815
2026-relevant notes	815
Related Commands	815
Additional Resources	815
dust	816
Overview	816
Syntax	816
Common Options	816
Examples with Explanations	817
Find big directories	817
Apparent size	817
Ignore noise	817
File types	817
Compare with du / ncdu	817
Min size filter	817
Scripts: stick to du	817
Notes / Pitfalls	818
2026-relevant notes	818
Related Commands	818
Additional Resources	818
parted	819
Overview	819
Syntax	819
Common Commands (interactive or CLI)	819
Safety	819
Examples with Explanations	820
Print table	820

GPT with one partition (scripted)	820
Interactive	820
Notes & Pitfalls	820
Related Commands	820
Additional Resources	821
wipefs	822
Overview	822
Syntax	822
Common Options	822
Safety	822
Examples with Explanations	823
Probe signatures	823
Erase all signatures	823
After wipe — new filesystem	823
Related Commands	823
Additional Resources	823
smartctl	824
Overview	824
Syntax	824
Common Options	824
Safety	824
Examples with Explanations	825
Health summary	825
Full report	825
Short self-test	825
Watch failure predictors	825
Scan devices	825
Related Commands	825
Additional Resources	826
pvs	827
Overview	827
Syntax	827
Common Options	827
Examples with Explanations	827
List physical volumes	827
Initialize a new PV (related workflow)	828
Free space across PVs	828
Notes & Pitfalls	828
Related Commands	828
Additional Resources	828
vgs	829
Overview	829
Syntax	829
Common Options	829

Examples with Explanations	829
List volume groups	829
Create VG from PVs (workflow)	829
Free space before creating an LV	830
Notes & Pitfalls	830
Related Commands	830
Additional Resources	830
lvs	831
Overview	831
Syntax	831
Common Options	831
Examples with Explanations	831
List LVs	831
Create and format an LV (workflow)	831
Grow an LV (then filesystem)	832
Snapshots (classic thick)	832
Understanding Output	832
Related Commands	832
Additional Resources	832
cryptsetup	833
Overview	833
Syntax	833
Common Commands	833
Safety	833
Examples with Explanations	834
Format and open (data disk)	834
Status and close	834
Add a second passphrase	834
Header backup	834
fstab / crypttab sketch	834
Notes & Pitfalls	834
Related Commands	835
Additional Resources	835
resize2fs	836
Overview	836
Syntax	836
Common Options	836
Safety	836
Examples with Explanations	837
Grow to fill LV after lvextend	837
Explicit size	837
Offline shrink (dangerous — backup first)	837
Notes & Pitfalls	837
Related Commands	837
Additional Resources	838

tune2fs	839
Overview	839
Syntax	839
Common Options	839
Safety	839
Examples with Explanations	840
Inspect	840
Label and UUID	840
Reserved space (big data disks)	840
Disable frequent forced fsck by mount count	840
Related Commands	840
Additional Resources	840
ncdu	841
Overview	841
Syntax	841
Common Options	841
Examples with Explanations	841
Scan root (one FS)	841
Home directory	842
Export for later / remote analysis	842
Notes & Pitfalls	842
Related Commands	842
Additional Resources	842
Terminals and Multiplexers	843
Intro	844
Intro	845
Commands in this part	845
Suggested starting points	845
Related parts	845
tmux	847
Overview	847
Syntax	847
Install (Ubuntu)	847
Prefix key	847
Common CLI commands	848
Key Use Cases	848
Examples with Explanations	849
Example: create, detach, reattach	849
Example: start detached job session	849
Example: nested SSH-safe habit	849
Example: splits for ops	849
Example: scripted window layout	849
Example: send keys to a session (automation)	850

Example: shared socket / pair-ish access	850
Example: minimal ~/.tmux.conf	850
Example: copy mode (scrollback)	850
Safety	851
Notes & Pitfalls	851
Related Commands	851
Additional Resources	851
screen	852
Overview	852
Syntax	852
Install (Ubuntu)	852
Prefix key	852
Common CLI options	853
Key Use Cases	853
Examples with Explanations	853
Example: create, detach, resume	853
Example: create-or-attach	854
Example: start detached command	854
Example: list and wipe zombies	854
Example: multi-attach	854
Example: send commands from outside	854
Example: log session output	854
Example: minimal ~/.screenrc	855
Example: vertical split note	855
Safety	855
Notes & Pitfalls	855
Related Commands	855
Additional Resources	856
byobu	857
Overview	857
Syntax	857
Common Options	857
Function Keys	857
Key Use Cases	858
Examples with Explanations	858
Example 1: Start Session	858
Example 2: Named Session	858
Example 3: List Sessions	858
Common Usage Patterns	859
Security Considerations	859
Related Commands	859
Additional Resources	859
Best Practices	859
Configuration	860
Troubleshooting	860

tmate	861
Overview	861
Syntax	861
Common usage patterns	861
Examples with Explanations	861
Start a session	861
Capture SSH URL for a colleague	862
Read-only vs read-write links	862
Config sketch	862
End session	862
Prefer local tmux when not sharing	862
Notes / Pitfalls	862
2026-relevant notes	863
Related Commands	863
Additional Resources	863
 abduco	 864
Overview	864
Syntax	864
Common Options	864
Examples with Explanations	865
Create / attach	865
Run a specific command	865
List and reattach	865
Detach	865
Read-only observe	865
Pair with dvtm	865
vs tmux	865
Notes / Pitfalls	866
2026-relevant notes	866
Related Commands	866
Additional Resources	866
 Text and Pipes	 867
 Intro	 868
 Intro	 869
Commands in this part	869
Suggested starting points	870
Related parts	870
 grep	 871
Overview	871
Syntax	871
Common Options	871
Key Use Cases	872

Examples with Explanations	872
Basic	872
Extended regex	872
Fixed string (fast, safe)	872
Recursive code search	872
Context	872
Files that match	872
Exit status in scripts	873
Only matching part	873
Invert	873
Understanding Output	873
Common Usage Patterns	873
Pipe from journalctl	873
Count HTTP status codes	873
Notes & Pitfalls	873
Related Commands	874
Additional Resources	874
sed	875
Overview	875
Syntax	875
Common Options	875
Common script commands	875
Safety	876
Key Use Cases	876
Examples with Explanations	876
Substitute first match per line	876
Global substitute	876
Extended regex	877
Delete matching lines	877
Print only rewritten matches	877
In-place with backup	877
Line addresses and ranges	877
Multiple expressions	877
Use match in replacement	878
Change delimiter when paths have slashes	878
Hold space: join two lines (advanced)	878
Dry-run pattern for risky edits	878
Understanding Output	878
Notes & Pitfalls	878
Related Commands	879
Additional Resources	879
awk	880
Overview	880
Syntax	880
Common Options	880
Key Use Cases	880

Examples with Explanations	881
Print columns	881
Filter rows	881
Sum and average a column	881
Change separators (CSV → TSV)	881
Last field and field count	881
Multiple patterns / counters	882
Build a frequency report	882
Extract IP from mixed lines	882
Skip header, process the rest	882
Script-friendly exit status	882
Field splitting on multi-character FS (gawk)	882
Understanding Output	883
Notes & Pitfalls	883
Related Commands	883
Additional Resources	883
cut	884
Overview	884
Syntax	884
Common Options	884
Field/Character Lists	884
Key Use Cases	885
Examples with Explanations	885
Example 1: Extract Fields	885
Example 2: Extract Characters	885
Example 3: Custom Delimiter	885
Working with Different Delimiters	885
Common Usage Patterns	885
Advanced Operations	886
Character vs Field Extraction	886
Performance Analysis	886
Related Commands	886
Additional Resources	887
Best Practices	887
Common Patterns	887
Integration Examples	887
Troubleshooting	888
sort	889
Overview	889
Syntax	889
Common Options	889
Key syntax (-k)	890
Examples with Explanations	890
Lexicographic sort	890
Numeric sort	890
Human-readable sizes	890

Version numbers	890
Sort by a column (whitespace)	890
CSV-ish second column (simple cases)	891
Unique lines	891
Frequency report (sort uniq -c sort)	891
Check if already sorted	891
Safe “sort in place”	891
Merge pre-sorted shards	891
Stable sort by key, preserve secondary order	891
NUL-safe with find	892
Randomize lines	892
Understanding Output	892
Notes & Pitfalls	892
Related Commands	892
Additional Resources	893
uniq	894
Overview	894
Syntax	894
Common Options	894
Examples with Explanations	894
Remove adjacent duplicates	894
The usual “unique lines in file” pattern	895
Count occurrences	895
Top IPs from a log (field extract → sort → uniq -c → sort)	895
Only lines that appear more than once	895
Only lines that appear exactly once	895
Case-insensitive uniqueness	895
Ignore leading fields when comparing	896
Compare only a prefix of each line	896
Show every line that is part of a duplicate group	896
NUL-separated path lists	896
Count total unique lines (quick)	896
Understanding Output	896
Notes & Pitfalls	897
Related Commands	897
Additional Resources	897
tr	898
Overview	898
Syntax	898
Common Options	898
Character Sets	898
Key Use Cases	899
Examples with Explanations	899
Example 1: Uppercase Conversion	899
Example 2: Delete Characters	899
Example 3: Replace Characters	899

Example 4: Squeeze Repeated Characters	899
Common Usage Patterns	900
Character Ranges	900
Advanced Usage	900
Text Processing	901
Performance Analysis	901
Related Commands	901
Best Practices	901
Data Cleaning	901
File Processing	902
Integration Examples	902
Scripting Applications	902
Special Characters	903
Troubleshooting	903
Security Applications	903
Performance Optimization	904
Real-world Examples	904
head	905
Overview	905
Syntax	905
Common Options	905
Key Use Cases	905
Examples with Explanations	906
Example 1: Default Usage	906
Example 2: Specific Line Count	906
Example 3: Multiple Files	906
Example 4: Byte Count	906
Common Usage Patterns	906
Advanced Usage	907
Performance Analysis	907
Related Commands	907
Best Practices	907
Integration Examples	907
Scripting Applications	908
Additional Examples	908
tail	909
Overview	909
Syntax	909
Common Options	909
Key Use Cases	909
Examples with Explanations	910
Example 1: Default Usage	910
Example 2: Follow Log File	910
Example 3: Specific Line Count	910
Example 4: Multiple Files	910
Follow Mode Options	910

Common Usage Patterns	911
Advanced Usage	911
Performance Analysis	911
Related Commands	911
Best Practices	912
Integration Examples	912
Log Rotation Handling	912
Scripting Applications	912
System Monitoring	913
Additional Examples	913

wc **914**

Overview	914
Syntax	914
Common Options	914
Default Output Format	914
Key Use Cases	915
Examples with Explanations	915
Example 1: Basic Count	915
Example 2: Lines Only	915
Example 3: Multiple Files	915
Understanding Counts	915
Common Usage Patterns	916
Advanced Usage	916
Pipeline Integration	916
Performance Analysis	917
Related Commands	917
Additional Resources	917
Best Practices	917
Scripting Examples	917
Character Encoding	918
Common Patterns	918
Integration Examples	918
Troubleshooting	919
Real-world Applications	919

tee **920**

Overview	920
Syntax	920
Common Options	920
Safety	920
Examples with Explanations	921
Save output and still see it	921
Append to a running log	921
Split to multiple files	921
Continue the pipeline after saving	921
Write files that need root (classic sudo tee)	921
Same pattern: append as root	921

Suppress terminal output but keep the file	922
Capture and process with jq later	922
Timestamped session log with a second process	922
Combine with process substitution (bash) for dual processing	922
Preserve exit status of the left side (bash pipefail)	922
Understanding Output	922
Notes & Pitfalls	923
Related Commands	923
Additional Resources	923
xargs	924
Overview	924
Syntax	924
Common Options	924
Safety	925
Examples with Explanations	925
Basic: echo words as args	925
Delete files listed by find (NUL-safe)	925
One invocation per item with placeholder	925
Limit batch size	925
Parallel compression	926
Show commands (-t) while learning	926
Skip empty input (GNU)	926
Lines as items (GNU -d)	926
Placeholders with multiple uses	926
From a file directly	926
Combine with git	927
Understanding Output	927
Notes & Pitfalls	927
Related Commands	927
Additional Resources	927
diff	928
Overview	928
Syntax	928
Common Options	928
Output Formats	928
Key Use Cases	929
Examples with Explanations	929
Example 1: Basic Comparison	929
Example 2: Unified Format	929
Example 3: Directory Comparison	929
Understanding Output	929
Common Usage Patterns	930
Advanced Options	930
Patch Creation	930
Performance Analysis	930
Related Commands	931

Additional Resources	931
Best Practices	931
Directory Comparison	931
Integration Examples	931
Scripting Applications	932
Special Cases	932
Troubleshooting	932
Output Redirection	933
Color Output	933
comm	934
Overview	934
Syntax	934
Common Options	934
Examples with Explanations	934
Basic three columns	934
Only in first / second / both	935
Users differences	935
Package set ops	935
Check order	935
Process substitution	935
Empty markers	935
Notes / Pitfalls	936
2026-relevant notes	936
Related Commands	936
Additional Resources	936
paste	937
Overview	937
Syntax	937
Common Options	937
Examples with Explanations	937
Side by side	937
Custom delimiter	938
Serialize lines of one file	938
Combine with cut/seq	938
Join data columns from commands	938
NUL-safe	938
Classic: make CSV from columns	938
Notes / Pitfalls	938
2026-relevant notes	939
Related Commands	939
Additional Resources	939
column	940
Overview	940
Syntax	940
Common Options	940

Examples with Explanations	940
Table-ize whitespace	940
Custom separators	941
Headers (newer)	941
Right-align numbers	941
JSON (when available)	941
Pretty fstab view	941
Pipelines	941
Notes / Pitfalls	941
2026-relevant notes	942
Related Commands	942
Additional Resources	942

jq **943**

Overview	943
Syntax	943
Common Options	943
Key Use Cases	943
Examples with Explanations	944
Pretty-print stdin	944
Extract a field	944
Map a list of objects to TSV	944
Filter array elements	944
Nested paths and optional keys	945
Update / reshape objects	945
Flatten and walk arrays	945
Build JSON from shell (safe)	945
Merge two JSON files	945
Exit status for CI	946
Compact logs / NDJSON	946
Understanding Output	946
Notes & Pitfalls	946
Related Commands	946
Additional Resources	947

pv **948**

Overview	948
Syntax	948
Common Options	948
Examples with Explanations	949
Copy with progress	949
dd-style disk image	949
Rate-limit a transfer	949
Multi-stage labeled bars	949
Related Commands	949
Additional Resources	949

Packages	950
Intro	951
Intro	952
Commands in this part	952
Suggested starting points	952
Related parts	953
apt	954
Overview	954
Syntax	954
Common Commands	954
Key Use Cases	954
Safety	955
Examples with Explanations	955
Refresh and upgrade	955
Install / remove	955
Search and inspect	955
Version pinning at install	956
Fix broken dependencies	956
Clean caches	956
Noninteractive (scripts/CI)	956
Simulate / dry-run	956
Which package owns a file?	957
Notes & Pitfalls	957
Related Commands	957
Additional Resources	957
dpkg	958
Overview	958
Syntax	958
Common Actions	958
Common Options	958
Key Use Cases	959
Examples with Explanations	959
Example 1: Install Package	959
Example 2: Remove Package	959
Example 3: List Files	959
Common Usage Patterns	960
Security Considerations	960
Related Commands	960
Additional Resources	960
Best Practices	960
Package States	961
Troubleshooting	961
dnf	962
Overview	962

Syntax	962
Common commands	962
Common options	963
Safety	963
Key Use Cases	963
Examples with Explanations	964
Example: refresh and update	964
Example: install and remove	964
Example: search and inspect	964
Example: which package owns a file	964
Example: repository visibility	964
Example: transaction history	965
Example: download only	965
Example: groups and modules	965
Example: security-focused updates	965
Example: clean caches	965
Example: scripted noninteractive	966
Notes & Pitfalls	966
Related Commands	966
Additional Resources	966

rpm 967

Overview	967
Syntax	967
Common Options	967
Query Options	967
Key Use Cases	968
Examples with Explanations	968
Example 1: Install Package	968
Example 2: Query Package	968
Example 3: Verify Package	968
Common Usage Patterns	969
Security Considerations	969
Related Commands	969
Additional Resources	969
Best Practices	969
Package Information	970
Troubleshooting	970

yum 971

Overview	971
Syntax	971
Common Commands	971
Common Options	971
Examples with Explanations	972
Everyday	972
Search and info	972
Repos	972

History / undo	972
Clean cache	973
Prefer dnf when present	973
Groups	973
Notes / Pitfalls	973
2026-relevant notes	973
Related Commands	973
Additional Resources	974
snap	975
Overview	975
Syntax	975
Common Commands	975
Safety	976
Key Use Cases	976
Examples with Explanations	976
Inventory and search	976
Install / remove (server-oriented)	976
Channels / tracks	977
Services and logs	977
Hold refreshes (ops)	977
Revisions / disk use	977
Notes & Pitfalls	977
Related Commands	978
Additional Resources	978
Services and Runtime	979
Intro	980
Intro	981
Commands in this part	981
Suggested starting points	982
Related parts	982
systemctl	983
Overview	983
Syntax	983
Common Commands	983
Key Use Cases	983
Safety	984
Examples with Explanations	984
Status and logs peek	984
Start / enable	984
Restart vs reload	984
Failed units	985
Script checks	985
After editing unit files	985

Drop-in override (preferred over editing vendor files)	985
Timers (cron alternative)	985
Dependencies and reverse deps	986
User services	986
Isolate targets (careful)	986
Understanding Output	986
Notes & Pitfalls	986
Related Commands	986
Additional Resources	987
journalctl	988
Overview	988
Syntax	988
Common Options	988
Key Use Cases	989
Examples with Explanations	989
Unit logs	989
Follow	989
Previous boot / crash triage	989
Kernel / dmesg-like	989
Time window	989
Priority	990
JSON for jq	990
Field matches	990
Disk usage and vacuum	990
Catalog / explain	990
Understanding Output	990
Notes & Pitfalls	991
Related Commands	991
Additional Resources	991
sysctl	992
Overview	992
Syntax	992
Common Options	992
Key Use Cases	992
Examples with Explanations	993
Read	993
Write (runtime)	993
Persist	993
One-liner recipes	993
Notes & Pitfalls	993
2026-relevant notes	994
Related Commands	994
Additional Resources	994
shutdown	995
Overview	995

Syntax	995
Time forms	995
Common options (shutdown)	995
systemd / loginctl cousins	996
Safety	996
Key Use Cases	996
Examples with Explanations	997
Example: power off now	997
Example: reboot now	997
Example: delayed reboot with message	997
Example: wall-clock schedule	997
Example: cancel	997
Example: message only (drill)	997
Example: systemd reboot with wall message	998
Example: check last reboots	998
Example: inhibit shutdown (awareness)	998
Example: forced paths (last resort)	998
Understanding Output	998
Notes & Pitfalls	998
Related Commands	999
Additional Resources	999
reboot	1000
Overview	1000
Syntax	1000
Common Options	1000
Examples with Explanations	1000
Normal reboot	1000
Scheduled	1001
With broadcast message	1001
Inhibitors	1001
Firmware setup (UEFI)	1001
After kernel install	1001
Containers / hosts	1001
Notes / Pitfalls	1001
2026-relevant notes	1002
Related Commands	1002
Additional Resources	1002
podman	1003
Overview	1003
Syntax	1003
Common Commands	1003
Key Use Cases	1003
Examples with Explanations	1004
Run	1004
Build	1004
Rootless tips	1004

systemd	1004
One-liner recipes	1005
Notes & Pitfalls	1005
2026-relevant notes	1005
Related Commands	1005
Additional Resources	1005
systemd-run	1006
Overview	1006
Syntax	1006
Common Options	1006
Safety	1006
Examples with Explanations	1007
One-shot with wait	1007
CPU/memory limits	1007
Transient timer	1007
User scope	1007
Interactive shell in a scope	1007
Related Commands	1007
Additional Resources	1008
systemd-analyze	1009
Overview	1009
Syntax	1009
Common Subcommands	1009
Examples with Explanations	1009
Boot timings	1009
Plot	1010
Verify units after editing	1010
Calendar expressions (timers)	1010
Security profile (where available)	1010
Notes & Pitfalls	1010
Related Commands	1010
Additional Resources	1011
loginctl	1012
Overview	1012
Syntax	1012
Common Commands	1012
Examples with Explanations	1012
Who is logged in	1012
Session details	1013
Enable lingering for user services	1013
Terminate a stale session	1013
Notes & Pitfalls	1013
Related Commands	1013
Additional Resources	1013

modprobe	1014
Overview	1014
Syntax	1014
Common Options	1014
Safety	1014
Examples with Explanations	1015
Load / unload	1015
Dependencies	1015
Module parameters	1015
Blacklist (persist)	1015
List available modules	1015
Related Commands	1015
Additional Resources	1016
 Shell Commands	 1017
Intro	1018
 Intro	 1019
Commands in this part	1019
Suggested starting points	1019
Related parts	1019
 echo	 1021
Overview	1021
Syntax	1021
Common Options	1021
Examples with Explanations	1021
Print text	1021
No newline	1022
Escapes (bash)	1022
Redirect / append	1022
Pipe into other commands	1022
Show a path or command you are about to run	1022
Notes	1022
Related Commands	1023
Additional Resources	1023
 printf	 1024
Overview	1024
Syntax	1024
Common Format Pieces	1024
Examples with Explanations	1024
Safe newline print	1024
No trailing newline	1025
Columns	1025
Zero-pad numbers	1025
Interpret escapes in data	1025

Hex / bytes (light)	1025
Build a file	1025
Arbitrary strings that look like options	1026
Notes	1026
Related Commands	1026
Additional Resources	1026
history	1027
Overview	1027
Syntax	1027
Common Options	1027
Examples with Explanations	1028
Show recent commands	1028
Search history (interactive bash)	1028
Re-run by number / last command	1028
Append now (multi-session)	1028
Clear session history	1028
Delete one awkward line	1028
Where history is stored	1029
Notes	1029
Related Commands	1029
Additional Resources	1029
type / command	1030
Overview	1030
Syntax	1030
Common Options	1030
type (bash)	1030
command	1030
Examples with Explanations	1031
What is this name?	1031
Machine-readable kind	1031
Path only	1031
Run without alias/function	1031
Compare to which / command -v	1031
Debug “wrong binary”	1032
Notes	1032
Related Commands	1032
Additional Resources	1032
export	1033
Overview	1033
Syntax	1033
Common Options	1033
Examples with Explanations	1033
Set for children	1033
One-shot without permanent export	1034
Show exports	1034

Unexport / unset	1034
Common operator variables	1034
Persist for your user (pointer)	1034
Notes	1034
Related Commands	1035
Additional Resources	1035
time	1036
Overview	1036
Syntax	1036
Common Options	1036
Bash builtin	1036
GNU /usr/bin/time (package time)	1036
Examples with Explanations	1037
Basic timing	1037
Time a pipeline (bash builtin)	1037
GNU verbose	1037
Portable three lines	1037
Custom format	1037
Compare two approaches	1037
Bash TIMEFORMAT	1038
Notes	1038
Related Commands	1038
Additional Resources	1038
pushd / popd / dirs	1039
Overview	1039
Syntax	1039
Common Options	1039
dirs	1039
pushd / popd	1039
Examples with Explanations	1040
Jump and return	1040
Nested hops	1040
List the stack	1040
Swap two project roots	1040
Numbered rotation	1041
Clear	1041
Notes	1041
Related Commands	1041
Additional Resources	1041
Security	1042
Intro	1043
Intro	1044
Commands in this part	1044

Suggested starting points	1044
Related parts	1044
getenforce	1045
Overview	1045
Syntax	1045
Examples with Explanations	1045
Check mode	1045
Related status	1045
Notes & Pitfalls	1045
Related Commands	1046
Additional Resources	1046
setenforce	1047
Overview	1047
Syntax	1047
Safety	1047
Examples with Explanations	1047
Temporary permissive for debugging	1047
Persist mode (config — not setenforce)	1048
Related Commands	1048
Additional Resources	1048
restorecon	1049
Overview	1049
Syntax	1049
Common Options	1049
Examples with Explanations	1049
Fix a web tree after rsync/mv	1049
Dry-run	1049
Single file after manual copy	1050
Notes & Pitfalls	1050
Related Commands	1050
Additional Resources	1050
getcap / setcap	1051
Overview	1051
Syntax	1051
Examples with Explanations	1051
Inspect	1051
Allow binding privileged ports (example)	1051
Remove capabilities	1052
Safety	1052
Notes & Pitfalls	1052
Related Commands	1052
Additional Resources	1052

Appendices	1053
Intro	1054
Intro	1055
Commands in this part	1055
Suggested starting points	1055
Related parts	1055
Ubuntu Cheatsheet	1056
Ubuntu Cheatsheet	1057
System	1057
System Information	1057
System Monitoring and Management	1057
Running Commands	1057
Service Management	1057
Cron Jobs and Scheduling	1057
Files	1058
File Management	1058
Directory Navigation	1058
File Permissions and Ownership	1058
Searching and Finding	1058
Archiving and Compression	1058
Text Editing and Processing	1058
Packages	1059
Package Management (APT)	1059
Package Management (Snap)	1059
Users & Groups	1059
User Management	1059
Group Management	1059
Networking	1060
Network Configuration	1060
Netplan Configuration	1060
Firewall Management	1060
SSH and Remote Access	1060
LXD	1060
Basic Setup	1060
Creating Instances	1061
Managing Instances	1061
Accessing Instances	1061
Using Projects	1061
Ubuntu Pro	1061
Activating Ubuntu Pro	1061
Managing Services	1062
Extended Security Maintenance (ESM)	1062
Livepatch Service	1062
FIPS Mode	1062

Updating Configuration	1062
Editors	1063
Intro	1064
Overview	1065
Overview: Editing on Linux	1066
The three layers	1066
Why terminal editors still matter	1066
Editor families (quick map)	1067
Workspace tools (quick map)	1067
A sane default stack (2026)	1067
Learning order inside this book	1068
Practice goals	1068
What “deepen” means here	1068
Choosing an Editor	1069
Choosing an Editor	1070
Decision matrix	1070
Time investment vs power	1070
Modal vs modeless	1071
GUI vs TUI vs remote	1071
Compatibility with the rest of your stack	1071
Recommendation cheat sheet	1072
Anti-patterns	1072
Pick one for thirty days	1072
Terminal Workspaces	1073
Terminal Workspaces	1074
Problems workspaces solve	1074
Core concepts (shared vocabulary)	1074
Session	1074
Window / tab	1074
Pane	1075
Prefix key	1075
Detach / attach	1075
Choosing among multiplexers	1075
When not to nest	1076
Editor integration patterns	1076
Pattern A — editor full window	1076
Pattern B — editor + shell split	1077
Pattern C — agent herd (Herdr)	1077
Pattern D — remote box	1077

Copy mode and scrollbar	1077
Persistence expectations	1077
Minimal practice lab	1078
Keybindings to keep conflict-free	1078
Next chapters	1079

Vi 1080

Vi: The Original Visual Editor 1081

Why Learn Vi?	1081
Vi Modes	1081
1. Command Mode (Normal Mode)	1081
2. Insert Mode	1081
3. Command-Line Mode	1081
Basic Operations	1082
Starting Vi	1082
Exiting Vi	1082
Navigation Commands	1083
Basic Movement	1083
Advanced Navigation	1083
Text Editing Commands	1084
Inserting Text	1084
Deleting Text	1084
Changing Text	1084
Copying and Pasting	1085
Working with Multiple Files	1085
Search and Replace	1085
Basic Search	1085
Search and Replace	1086
Advanced Features	1086
Marks and Bookmarks	1086
Registers	1086
Macros	1087
Configuration	1087
Vi Configuration File (.exrc)	1087
Environment Variables	1087
Real-World Scenarios	1088
Scenario 1: Emergency Linux System Recovery	1088
Scenario 2: Linux Log File Analysis	1088
Scenario 3: Linux System Configuration	1089
Vi vs Modern Editors	1090
Advantages of Vi	1090
Limitations	1090
Tips and Best Practices	1090
Essential Tips	1090
Common Mistakes	1091
Productivity Shortcuts	1091

Troubleshooting	1091
Common Issues	1091
Recovery Options	1092

Vim 1093

Vim: Vi Improved 1094

Why Choose Vim?	1094
Installation	1094
Linux Distributions	1094
Building from Source	1095
Vim Modes	1095
1. Normal Mode	1095
2. Insert Mode	1095
3. Visual Mode	1095
4. Command-Line Mode	1095
5. Replace Mode	1096
Enhanced Navigation	1096
Advanced Movement	1096
Window and Tab Management	1096
Advanced Text Editing	1097
Visual Mode Operations	1097
Advanced Copy/Paste	1098
Text Objects	1098
Configuration and Customization	1099
Vimrc Configuration	1099
Advanced Vimrc Features	1101
Plugin Management	1102
Using vim-plug	1102
Installing Plugins	1103
Programming with Vim	1104
Code Navigation	1104
Code Completion	1104
Debugging Integration	1104
Real-World Development Scenarios	1105
Scenario 1: Python Development Setup	1105
Scenario 2: Web Development Workflow	1105
Scenario 3: Linux System Administration	1106
Advanced Vim Techniques	1106
Macros and Automation	1106
Regular Expressions	1107
Custom Commands	1107
Performance Optimization	1107
Vim Performance Tips	1107
Large File Handling	1108
Troubleshooting Common Issues	1108
Performance Problems	1108

Plugin Conflicts	1108
Recovery and Backup	1109
Neovim	1110
Neovim: The Future of Vim	1111
Why Choose Neovim?	1111
Installation	1111
Package Managers	1111
From Source	1112
AppImage (Portable)	1112
Configuration Structure	1112
Basic Lua Configuration	1113
init.lua Structure	1113
Plugin Management with Packer	1114
Installing Packer	1114
Packer Configuration	1114
Loading Plugins in init.lua	1117
Language Server Protocol (LSP) Setup	1117
LSP Configuration	1117
Installing Language Servers	1119
Tree-sitter Configuration	1119
Tree-sitter Setup	1119
Advanced Neovim Features	1120
Telescope Configuration	1120
Terminal Integration	1121
Real-World Development Setups	1122
Python Development	1122
JavaScript/TypeScript Development	1123
Go Development	1124
Debugging with DAP	1124
DAP Configuration	1124
Performance Optimization	1125
Startup Optimization	1125
Migration from Vim	1126
Compatibility Layer	1126
Troubleshooting	1127
Common Issues	1127
Nano	1128
Nano: The User-Friendly Terminal Editor	1129
Why Choose Nano?	1129
Installation	1129
Most Linux Distributions (Pre-installed)	1129
Additional Linux Distributions	1130

Basic Usage	1130
Starting Nano	1130
Interface Overview	1130
Essential Commands	1131
File Operations	1131
Navigation	1131
Text Editing	1132
Search and Replace	1132
Configuration	1133
Global Configuration	1133
Sample .nanorc Configuration	1133
Syntax Highlighting	1135
Custom Syntax Highlighting	1136
Advanced Features	1136
Multiple Buffers	1136
Spell Checking	1136
Backup Files	1137
Mouse Support	1137
Real-World Usage Scenarios	1137
Scenario 1: Quick Configuration File Edit	1137
Scenario 2: System Log Analysis	1138
Scenario 3: Script Development	1138
Scenario 4: Crontab Editing	1139
Command Line Options	1139
Useful Nano Options	1139
Environment Variables	1140
Tips and Best Practices	1140
Productivity Tips	1140
Common Shortcuts Summary	1140
Customization for Different Use Cases	1141
Troubleshooting	1142
Common Issues	1142
Recovery Options	1143

Emacs 1144

Emacs: The Extensible Editor 1145

Why Choose Emacs?	1145
Installation	1145
Linux Distributions	1145
Additional Linux Distributions	1146
Building from Source	1146
Basic Concepts	1146
Key Notation	1146
Buffers, Windows, and Frames	1146
Major and Minor Modes	1147

Essential Commands	1147
Starting and Exiting	1147
File Operations	1147
Buffer Management	1148
Window Management	1148
Navigation and Editing	1149
Basic Movement	1149
Advanced Navigation	1149
Text Editing	1150
Selection (Regions)	1150
Search and Replace	1151
Basic Search	1151
Replace Operations	1151
Advanced Search	1151
Configuration	1152
Init File	1152
Package Management	1154
Popular Packages	1154
Programming with Emacs	1155
Language-Specific Modes	1155
Code Navigation	1156
Debugging	1157
Org Mode	1157
Basic Org Syntax	1157
Org Mode Commands	1158
Org Configuration	1158
Real-World Development Scenarios	1159
Scenario 1: Python Development Setup	1159
Scenario 2: Web Development Workflow	1160
Scenario 3: System Administration	1160
Advanced Emacs Features	1161
Macros	1161
Registers	1161
Direc (Directory Editor)	1162
Shell Integration	1162
Customization and Themes	1162
Custom Themes	1162
Custom Functions	1163
Performance Optimization	1164
Startup Optimization	1164
Large File Handling	1164
Troubleshooting	1164
Common Issues	1164
Recovery	1165

LazyVim	1166
LazyVim: Modern Neovim Configuration	1167
What is LazyVim?	1167
Why Choose LazyVim?	1167
Installation	1167
Prerequisites	1167
Required Dependencies	1168
Installation Steps	1168
Initial Setup and Configuration	1169
First Launch	1169
Basic Configuration Structure	1169
Basic Options Configuration	1169
Custom Keymaps	1171
Auto Commands	1172
Plugin Configuration	1174
Adding Custom Plugins	1174
Language-Specific Configurations	1175
Essential LazyVim Features	1177
File Explorer (Neo-tree)	1177
Fuzzy Finding (Telescope)	1177
LSP Features	1178
Git Integration (Lazygit)	1179
Terminal Integration	1179
Real-World Development Setups	1180
Python Development	1180
Web Development	1182
Go Development	1184
Customization Examples	1186
Custom Colorscheme	1186
Custom Dashboard	1187
Performance Optimization	1189
Startup Optimization	1189
Memory Optimization	1190
Troubleshooting	1192
Common Issues	1192
Reset Configuration	1192
Performance Issues	1193
 Helix	 1194
Helix: A Modern Modal Editor	1195
What is Helix?	1195
Why Choose Helix?	1195
Installation	1195
Package Managers	1195
Building from Source	1196

Post-Installation Setup	1196
Basic Concepts	1197
Selection-First Editing	1197
Multiple Selections	1197
Modes	1197
Essential Commands	1198
Starting Helix	1198
Basic Navigation	1198
Text Selection	1199
Editing Commands	1199
Search and Replace	1200
File and Buffer Management	1201
File Operations	1201
File Picker and Fuzzy Finding	1201
Configuration	1201
Configuration File	1201
Language Configuration	1204
Themes	1205
Custom Theme	1206
Real-World Usage Scenarios	1207
Scenario 1: Python Development	1207
Scenario 2: Web Development	1207
Scenario 3: Configuration File Editing	1208
Scenario 4: Log File Analysis	1208
Advanced Features	1209
Multiple Selections Mastery	1209
Text Objects	1209
Tree-sitter Navigation	1210
LSP Features	1210
Customization and Scripting	1211
Custom Commands	1211
Integration with External Tools	1211
Shell Integration	1211
Performance and Optimization	1212
Large File Handling	1212
Memory Usage	1212
Troubleshooting	1213
Common Issues	1213
Debug Mode	1213
Reset Configuration	1213
Migration from Other Editors	1213
From Vim/Neovim	1213
From VSCode	1214
From Kakoune	1214

Micro	1215
Micro: A Modern Terminal Editor	1216
What is Micro?	1216
Why Choose Micro?	1216
Installation	1216
Package Managers	1216
Direct Download	1217
From Source	1217
Basic Usage	1218
Starting Micro	1218
Interface Overview	1218
Essential Keybindings	1219
File Operations	1219
Navigation	1219
Text Editing	1219
Multiple Cursors	1220
Search and Replace	1220
Configuration	1220
Settings File	1220
Common Configuration Changes	1222
Keybinding Customization	1222
Color Schemes	1223
Custom Color Scheme	1223
Plugin System	1224
Plugin Management	1224
Popular Plugins	1224
Plugin Configuration	1225
Real-World Usage Scenarios	1225
Scenario 1: Linux System Configuration	1225
Scenario 2: Python Development	1226
Scenario 3: Log File Analysis	1227
Scenario 4: Web Development	1227
Advanced Features	1228
Command Mode	1228
Macros	1229
Bookmarks	1229
Splits and Tabs	1229
Integration with Development Tools	1230
Git Integration	1230
Language Server Integration	1230
Build System Integration	1230
Customization Examples	1231
Custom Status Line	1231
Custom Key Bindings for Productivity	1231
File Type Specific Settings	1231
Performance and Optimization	1232
Large File Handling	1232

Memory Usage	1232
Troubleshooting	1233
Common Issues	1233
Debug Mode	1233
Reset Configuration	1233
Migration from Other Editors	1233
From Nano	1233
From GUI Editors	1234
From Vim	1234

Kakoune 1235

Kakoune: The Selection-Based Editor 1236

What is Kakoune?	1236
Why Choose Kakoune?	1236
Installation	1236
Package Managers	1236
Building from Source	1237
Development Version	1237
Basic Concepts	1238
Selection-First Philosophy	1238
Multiple Selections	1238
Modes	1238
Essential Commands	1238
Starting Kakoune	1238
Basic Navigation	1239
Text Selection	1240
Multiple Selections	1240
Editing Commands	1241
Search and Replace	1241
File and Buffer Management	1242
Buffer Operations	1242
File Operations	1242
Window Management	1243
Configuration	1243
Configuration File	1243
Plugin Management	1244
Popular Plugins	1245
Real-World Usage Scenarios	1246
Scenario 1: Refactoring Variable Names	1246
Scenario 2: Multiple Line Editing	1247
Scenario 3: Complex Text Manipulation	1247
Scenario 4: Log File Analysis	1248
Advanced Features	1248
Hooks and Automation	1248
Custom Commands	1249
Text Objects	1249

Macros and Repetition	1250
Language-Specific Configurations	1250
Python Development	1250
Web Development	1251
Go Development	1252
Integration with External Tools	1252
Git Integration	1252
Build System Integration	1253
Terminal Integration	1254
Performance and Optimization	1254
Large File Handling	1254
Memory Optimization	1255
Troubleshooting	1255
Common Issues	1255
Session Management	1255
Configuration Issues	1256
Modern Tools	1257
Modern Tools	1258
Modern Linux Development Tools	1259
Why Modern Tools on Linux?	1259
Suggested “editor kit” install	1259
Terminal Multiplexers (map)	1259
GNU screen (legacy multiplexer)	1260
Terminal-Based Development Tools	1260
Zellij: Modern Terminal Multiplexer	1260
Starship: Cross-Shell Prompt	1261
Zoxide: Smart Directory Navigation	1264
Eza: Modern ls Replacement	1265
Bat: Better Cat	1266
Ripgrep: Fast Text Search	1266
Fzf: Fuzzy Finder	1267
Lazygit: Terminal Git UI	1268
Yazi: Terminal File Manager (modern)	1269
Fd: Better Find	1269
Delta: Better Git Diff	1270
File Management Tools	1271
Broot: Interactive Directory Navigation	1271
Nnn: Terminal File Manager	1272
Development Environment Tools	1273
Direnv: Environment Management	1273
Mise: Runtime Version Manager	1274
Just: Command Runner	1276
Text Processing Tools	1278
Jq: JSON Processor	1278

Yq: YAML Processor	1278
Sd: Sed Alternative	1279
Monitoring and System Tools	1280
Htop: Process Viewer	1280
Bottom: System Monitor	1281
Procs: Process Information	1281
Linux-Specific Integration	1282
Systemd Integration	1282
Linux Distribution Package Management	1283
Linux Desktop Integration	1284
Integration Examples	1285
Complete Development Setup	1285
Project Template with Modern Tools	1286
Linux-Specific Troubleshooting	1288
Permission Issues	1288
Package Manager Conflicts	1288
Performance Optimization for Linux	1289
Container Integration	1289
Putting it together with editors	1290

Tmux 1291

tmux 1292

tmux: Terminal Multiplexer 1293

Why tmux	1293
Installation	1293
Mental model	1294
Quick start lab	1294
Essential keybindings	1295
Sessions	1295
Windows	1295
Panes	1295
Copy mode (scrollback)	1296
Command-line interface	1296
Send keys and automation	1297
Configuration (~/.tmux.conf)	1297
Sensible starter config	1297
Nested tmux (SSH jump hosts)	1298
Nested editors and key conflicts	1299
SSH workflows	1299
Keepalive	1299
Plugins (optional)	1299
Pairing with editors	1300
Neovim / Vim	1300
Helix	1300
Emacs	1300

Nano / Micro	1301
Comparison: tmux vs screen vs Zellij vs Herdr	1301
Troubleshooting	1301
Cheatsheet card	1301
Practice checklist	1302
Herdr	1303
Herdr	1304
Herdr: Agent-Native Terminal Workspaces	1305
Why Herdr exists	1305
Install	1306
Quick install (Linux / macOS)	1306
Package managers	1306
Manual binary	1306
Channels	1306
Concepts	1307
Quick start	1307
Keyboard and mouse	1308
CLI automation	1308
Working with editors	1309
\$EDITOR still matters	1310
Herdr vs tmux vs Zellij	1310
Agents and detection	1310
Configuration sketch	1311
Plugins	1311
Session state and persistence	1311
Security notes	1312
Troubleshooting	1312
Practice lab	1312
When to use what	1312
Further reading	1313
Zed	1314
Zed	1315
Zed: Fast Native Code Editor	1316
Why Zed on Linux	1316
Install	1316
Wait flag for \$EDITOR	1317
Core UI map	1317
Settings and config	1317
Themes	1318
Vim mode	1318
Language servers and extensions	1318

Projects and workspaces	1319
Pairing with tmux and Herdr	1319
A. Local Zed + terminal multiplexer for agents	1319
B. All in terminal (no Zed remote)	1319
C. Remote code	1319
Collaboration	1320
Git workflow	1320
Comparison: Zed vs neighbors	1320
Migration tips	1320
From VS Code	1320
From Neovim	1321
From Helix	1321
Troubleshooting	1321
Practice checklist	1321
When to choose Zed	1321
Workflows	1323
Workflows	1324
Workflows: Editors + Workspaces + Tools	1325
Stack templates	1325
1) Server / sysadmin (minimal)	1325
2) Terminal IDE (Neovim)	1325
3) Selection-first modal (Helix)	1326
4) Friendly modeless (Micro)	1326
5) Desktop GUI + agents (Zed + Herdr)	1326
6) Multi-agent feature work (Herdr-centric)	1326
7) Emacs as environment	1327
Cross-cutting recipes	1327
Open search hits in your editor	1327
Git commit with correct blocking editor	1327
kubectrl / sudoedit	1328
One-shot session script (tmux)	1328
Agent + editor discipline	1328
Keybinding peace treaty	1328
Dotfiles layout (suggested)	1329
Choosing under pressure	1329
Practice: build your “default”	1329
See also	1329
Scripting	1331
Bash	1332

Intro to Shells	1333
Shell vs Terminal: Understanding the Difference	1334
What is a Terminal?	1334
Common Terminal Emulators	1334
What is a Shell?	1335
The Relationship	1335
Practical Examples	1335
Example 1: Multiple Shells in One Terminal	1335
Example 2: Same Shell in Different Terminals	1336
Example 3: Remote Shells	1336
Terminal Features vs Shell Features	1336
Terminal Features:	1336
Shell Features:	1336
Configuration Files	1336
Terminal Configuration:	1336
Shell Configuration:	1337
Choosing Terminal and Shell Combinations	1337
For Development:	1337
For System Administration:	1337
For Beginners:	1337
Common Misconceptions	1337
Misconception 1: “I’m using Terminal”	1337
Misconception 2: “Bash is a terminal”	1337
Misconception 3: “All terminals are the same”	1338
Practical Tips	1338
Check what terminal you’re using:	1338
Check what shell you’re using:	1338
Launch different shells:	1338
Summary	1338
Types of Shells	1340
Common Shell Types	1340
1. Bourne Shell (sh)	1340
2. Bash (Bourne Again Shell)	1340
3. Zsh (Z Shell)	1341
4. Fish (Friendly Interactive Shell)	1341
5. Dash (Debian Almquist Shell)	1341
6. Tcsh (TENEX C Shell)	1341
7. Ksh (Korn Shell)	1342
Checking Available Shells	1342
View all available shells:	1342
Check your current shell:	1342
Check shell being used by a script:	1342
Shell Comparison	1342
Choosing the Right Shell	1343
For Scripting:	1343
For Interactive Use:	1343

For System Administration:	1343
Switching Between Shells	1343
Temporarily switch:	1343
Change default shell:	1344
Shell Compatibility	1344
What is a Shell?	1345
The Role of a Shell	1345
How Shells Work	1345
Interactive vs Non-Interactive Shells	1345
Interactive Shell	1345
Non-Interactive Shell	1346
Login vs Non-Login Shells	1346
Login Shell	1346
Non-Login Shell	1346
Shell Capabilities	1346
Example: Basic Shell Interaction	1346
Scripting	1348
Your First Shell Script	1349
Step 1: Create Your First Script	1349
Using a Text Editor	1349
Write the Script	1349
Save the File	1349
Step 2: Understanding the Script	1350
Shebang Line	1350
Comments	1350
Commands	1350
Command Substitution	1350
Step 3: Make the Script Executable	1350
Understanding Permissions	1351
Step 4: Run Your Script	1351
Method 1: Direct Execution	1351
Method 2: Using bash Command	1351
Method 3: Using sh Command	1352
Method 4: Full Path	1352
Step 5: Enhance Your Script	1352
Step 6: Adding Error Handling	1353
Step 7: Script with Functions	1354
Common Beginner Mistakes	1355
1. Forgetting the Shebang	1355
2. Not Making Script Executable	1355
3. Wrong Path in Shebang	1355
4. Spaces in Variable Assignment	1355
Best Practices for Beginners	1356
1. Always Use Shebang	1356

2. Add Comments	1356
3. Use Meaningful Names	1356
4. Quote Variables	1356
5. Check for Errors	1357
Testing Your Scripts	1357
1. Syntax Check	1357
2. Debug Mode	1357
3. Use shellcheck	1357
Next Steps	1357
Script Execution Methods	1358
Execution Methods Overview	1358
Method 1: Direct Execution	1358
Prerequisites	1358
Execution	1358
Example Script	1359
Test It	1359
Method 2: Using Shell Interpreter	1359
Bash Interpreter	1359
Other Interpreters	1359
Advantages	1360
Example	1360
Method 3: Source (Dot) Command	1360
Syntax	1360
Key Characteristics	1360
Example Script	1361
Test Sourcing	1361
Method 4: Absolute Path Execution	1361
Examples	1362
Use Cases	1362
Method 5: Command Substitution	1362
Syntax	1362
Example	1362
Execution Context and Environment	1363
Process Creation	1363
Environment Inheritance	1363
Test Environment	1363
Script Arguments and Parameters	1364
Passing Arguments	1364
Accessing Arguments	1364
Test Arguments	1364
Exit Status and Error Handling	1364
Exit Status	1364
Check Exit Status	1365
Advanced Execution Techniques	1365
Background Execution	1365
Conditional Execution	1365
Piping Script Output	1365

Debugging Script Execution	1366
Debug Mode	1366
Verbose Mode	1366
Syntax Check	1366
Combined Options	1366
Security Considerations	1367
File Permissions	1367
PATH Security	1367
Input Validation	1367
Best Practices	1367
1. Always Use Shebang	1367
2. Set Error Handling	1368
3. Use Appropriate Execution Method	1368
4. Handle Arguments Properly	1368
5. Provide Clear Output	1368
What is Shell Scripting?	1369
Definition and Purpose	1369
Key Benefits:	1369
When to Use Shell Scripts	1369
Perfect for:	1369
Not Ideal for:	1369
Types of Shell Scripts	1370
1. Simple Command Sequences	1370
2. Interactive Scripts	1370
3. System Administration Scripts	1370
4. Data Processing Scripts	1370
Shell Script Components	1371
1. Shebang Line	1371
2. Comments	1371
3. Variables	1371
4. Commands	1371
5. Control Structures	1371
Script Execution Methods	1372
1. Make Executable and Run	1372
2. Run with Shell Interpreter	1372
3. Source the Script	1372
Script Structure Best Practices	1372
Basic Template:	1372
Common Shell Scripting Patterns	1373
1. Error Handling	1373
2. Argument Processing	1373
3. Configuration Files	1374
Shell Scripting vs Other Languages	1374
Shell Scripts Excel At:	1374
Other Languages Better For:	1374
Development Environment	1374
Essential Tools:	1374

Useful Commands:	1375
Real-World Example	1375
Bash Features	1376
Bash Environment	1377
Environment Variables	1377
Viewing Environment Variables	1377
Common Environment Variables	1378
Setting Environment Variables	1379
Unsetting Variables	1379
Shell Variables vs Environment Variables	1379
Shell Variables	1379
Environment Variables	1380
Special Variables	1380
Positional Parameters	1380
Process Variables	1381
Test Special Variables	1381
Configuration Files	1381
Login Shell Files (in order)	1381
Non-Login Interactive Shell Files	1381
Non-Interactive Shell Files	1381
Logout Files	1381
Configuration File Examples	1382
Prompt Customization	1383
Basic Prompt Variables	1383
Colorized Prompt	1383
Advanced Prompt with Git Status	1384
PATH Management	1384
Viewing PATH	1384
Modifying PATH	1384
PATH Best Practices	1385
Shell Options and Settings	1385
Set Options	1385
Shopt Options	1385
Environment Management Scripts	1386
Environment Setup Script	1386
Environment Backup and Restore	1387
Debugging Environment Issues	1387
Common Environment Problems	1387
Environment Diagnostic Script	1387
Bash Features	1389
Interactive Features	1389
Command History	1389
History Configuration	1389
Tab Completion	1390

Custom Completion	1390
Command Line Editing	1390
Job Control	1391
Background Jobs	1391
Job Management	1391
Process Control	1392
Aliases and Functions	1392
Aliases	1392
Functions	1393
Variable Features	1394
Variable Types	1394
Parameter Expansion	1394
Default Values	1395
Pattern Matching and Globbing	1395
Basic Wildcards	1395
Extended Globbing	1395
Globstar (Bash 4+)	1396
Brace Expansion	1396
Sequence Generation	1396
String Expansion	1397
Process Substitution	1397
Input Substitution	1397
Output Substitution	1397
Command Substitution	1398
Modern Syntax	1398
Legacy Syntax	1398
Nested Substitution	1398
Arithmetic Operations	1398
Arithmetic Expansion	1398
Arithmetic Commands	1399
Advanced Arithmetic	1399
Conditional Expressions	1399
Test Command	1399
Extended Test	1400
I/O Redirection	1400
Basic Redirection	1400
Advanced Redirection	1400
Shell Options	1401
Set Options	1401
Shopt Options	1401
What is Bash?	1402
History and Background	1402
The Evolution	1402
Why “Bourne Again”?	1402
Bash Characteristics	1402
POSIX Compliance	1402
GNU Project	1403

Bash Versions	1403
Check Your Bash Version	1403
Major Version Differences	1403
Version-Specific Features	1404
Bash vs Other Shells	1404
Bash vs Bourne Shell (sh)	1404
Bash vs Zsh	1404
Bash vs Fish	1404
Where Bash is Used	1405
Default Shell	1405
System Scripts	1405
Development and DevOps	1405
Bash Capabilities	1406
Interactive Features	1406
Programming Features	1406
Advanced Features	1407
Bash Configuration	1407
Configuration Files	1407
Customization Example	1408
Bash Built-in Commands	1408
Common Built-ins	1408
Bash Scripting Advantages	1409
1. Ubiquity	1409
2. Integration	1409
3. Learning Curve	1409
4. Power and Flexibility	1409
When to Use Bash	1410
Perfect For:	1410
Consider Alternatives For:	1410
Getting Started with Bash	1410
Check if Bash is Available	1410
Start Using Bash	1410
First Bash Script	1410

Variables 1412

Variable Basics 1413

What are Variables?	1413
Variable Declaration and Assignment	1413
Basic Syntax	1413
Important Rules	1413
Common Mistakes	1414
Variable Naming Conventions	1414
Valid Variable Names	1414
Invalid Variable Names	1414
Best Practices	1414

Accessing Variable Values	1415
Basic Access	1415
When to Use Braces	1415
Variable Types and Behavior	1416
String Variables	1416
Numeric Variables	1416
Boolean-like Variables	1416
Variable Scope	1417
Global Variables	1417
Local Variables	1417
Reading User Input	1417
Basic Input	1417
Input with Prompt	1418
Multiple Variables	1418
Reading into Array	1418
Silent Input (Passwords)	1418
Variable Assignment Methods	1418
Direct Assignment	1418
Command Substitution	1419
Reading from Files	1419
Arithmetic Assignment	1419
Variable Manipulation	1420
String Length	1420
Substring Extraction	1420
Case Conversion	1420
Pattern Replacement	1420
Default Values and Error Handling	1421
Default Values	1421
Error on Unset Variables	1421
Practical Example	1421
Variable Arrays (Preview)	1422
Simple Array	1422
Best Practices	1422
1. Always Quote Variables	1422
2. Use Meaningful Names	1422
3. Initialize Variables	1423
4. Use readonly for Constants	1423
5. Validate Input	1423
Common Variable Patterns	1424
Configuration Variables	1424
Temporary Variables	1424

Control **1425**

Conditional Statements **1426**

The if Statement	1426
Basic Syntax	1426

Simple Example	1426
if-else Statement	1426
if-elif-else Statement	1427
Test Conditions	1427
Numeric Comparisons	1427
String Comparisons	1428
File Tests	1429
Advanced Test Constructs	1430
Double Brackets [[]]	1430
Arithmetic Evaluation (())	1430
Logical Operators	1431
AND Operator (&&)	1431
OR Operator ()	1431
NOT Operator (!)	1432
Practical Examples	1432
File Backup Script	1432
System Health Check	1433
User Input Validation	1434
Nested Conditionals	1435
Short-Circuit Evaluation	1435
Common Pitfalls and Best Practices	1436
1. Always Quote Variables	1436
2. Use [[]] for String Operations	1436
3. Handle Empty Variables	1437
4. Use Meaningful Conditions	1437
5. Exit Codes for Error Handling	1437

Loops 1438

for Loops	1438
Basic for Loop Syntax	1438
Simple Examples	1438
Looping Through Files	1438
C-style for Loop	1439
Using Brace Expansion	1439
Looping Through Arrays	1440
while Loops	1440
Basic while Loop	1440
Reading File Line by Line	1441
Menu System with while Loop	1441
Monitoring with while Loop	1442
until Loops	1442
Basic until Loop	1443
Waiting for a Service	1443
Waiting for File	1443
Loop Control Statements	1444
break Statement	1444
continue Statement	1444
Nested Loops with break and continue	1444

Practical Loop Examples	1445
Batch File Processing	1445
Log Analysis	1446
System Backup Script	1446
User Management Script	1447
Loop Performance Tips	1448
Avoid Unnecessary Command Substitution	1448
Use Built-in Commands When Possible	1449
Process Files Efficiently	1449
Common Loop Patterns	1449
Retry Pattern	1449
Progress Indicator	1450

Functions 1451

Function Basics 1452

What are Functions?	1452
Function Syntax	1452
Method 1: Using the <code>function</code> keyword	1452
Method 2: POSIX-compatible syntax (preferred)	1452
Basic Function Examples	1453
Simple Function	1453
Function with Commands	1453
Function Parameters	1453
Accessing Parameters	1453
Multiple Parameters	1454
Parameter Variables	1454
Return Values	1454
Return Exit Status	1455
Return Values via Echo	1455
Local Variables	1455
Without Local Variables (Global)	1456
With Local Variables	1456
Best Practice Example	1456
Practical Function Examples	1457
File Operations Function	1457
System Information Function	1458
String Processing Function	1458
Function Libraries	1459
Create a Library File (<code>utils.sh</code>)	1459
Use the Library in Your Script	1460
Function Best Practices	1461
1. Use Descriptive Names	1461
2. Validate Input Parameters	1461
3. Use Local Variables	1462
4. Provide Usage Information	1462
5. Handle Errors Gracefully	1463

Common Function Patterns	1463
Configuration Function	1463
Cleanup Function	1463
Menu Function	1464
Misc	1465
Loop from 1 to 5	1466
Loop from 1 to 10, incrementing by 2	1467
Loop over a list of strings	1468
Define an array	1469
Loop over the array	1470
Loop from 1 to 5	1471
Loop through an array and break when a specific value is found	1472
Loop through a range of numbers and skip multiples of 3	1473
Workbook	1474
Basic Bash Examples	1475
Hello World Variations	1475
Simple Hello World	1475
Hello World with Variables	1475
Interactive Hello World	1475
Hello World with Date	1475
Variable Examples	1476
Basic Variable Operations	1476
Environment Variables	1476
Variable Default Values	1476
Variable Length and Manipulation	1477
Command Line Arguments	1477
Basic Argument Handling	1477
Argument Validation	1477
Processing Multiple Arguments	1478
Simple Calculations	1478
Basic Arithmetic	1478
Calculator Script	1478
Temperature Converter	1479
File Operations	1479
File Information	1479
File Backup	1480
Directory Listing	1480

Text Processing	1481
Word Count	1481
Text Search	1481
Line Numbering	1482
System Information	1482
System Status	1482
Process Monitor	1483
Network Information	1483
User Input Examples	1484
Menu System	1484
User Registration	1484
Quiz Game	1485
String Manipulation	1486
String Operations	1486
Palindrome Checker	1487
Password Generator	1487
Array Examples	1488
Basic Array Operations	1488
Shopping List	1488
Grade Calculator	1490
Intermediate Bash Examples	1492
Function Examples	1492
Library of Utility Functions	1492
Configuration Manager	1494
Database Backup Script	1497
File Processing Examples	1501
Log Analyzer	1501
File Organizer	1503
CSV Processor	1507
System Administration Examples	1513
System Monitor	1513
Advanced Bash Examples	1522
Process Management and IPC	1522
Process Pool Manager	1522
Network Programming	1526
HTTP Server in Bash	1526
Network Scanner	1530
Data Processing and Analysis	1534
JSON Processor	1534
Data Aggregator	1537
Security and Encryption	1541
Password Manager	1541
Automation Examples	1548
System Administration Automation	1548
Automated Backup System	1548

System Health Monitor	1556
Environment Provisioning Script	1575
Web Scraping and API Automation	1584
Web Scraper	1584
Utility Scripts	1595
File and Directory Utilities	1595
Duplicate File Finder	1595
Directory Synchronizer	1603
System Maintenance Utilities	1613
Log Rotator	1613

Introduction

Linux

A practical library for **Linux on the command line**, organized as three parts under one book:

Linux	
Commands	→ topics (files, processes, networking, ...) → individual commands
Editors	→ vi, Neovim, tmux, Herdr, ... → chapter pages
Scripting	
Bash	→ shells, scripting, variables, control, workbook

This landing page restores the **Commands** orientation (scope, page template, parts map) from the former standalone *Linux-Commands* book, and points you into Editors and Bash as well.

Commands — scope

A practical, **Ubuntu-first** encyclopedia of **general Linux CLI** and **server-side** commands for administration, scripting, and troubleshooting on headless and SSH-managed systems.

Commands stays on the terminal and the OS. It does **not** cover desktop/GUI applications, graphical package stores, or printer/desktop workflows.

Each Commands chapter covers **one command** (or a tight family): syntax, important options, realistic examples, output notes, safety tips where it matters, and related tools.

In scope	Out of scope
Core userland (ls , find , pipes, ...)	Desktop apps and GUI installers
Networking, SSH, firewalls, capture	Display servers (X11/Wayland), window managers
systemd, journals, packages	Flatpak / GUI app stores
(apt / dnf)	
Storage, LVM, LUKS, disk health	CUPS client printing queues
Processes, monitoring, scheduling	Desktop session “app” management
Terminal multiplexers (tmux , ...)	Browser-based or GUI-only tools
SELinux/capabilities as server	End-user desktop customization
concerns	

Editors and Bash live in the other parts of **this** book (they used to be sibling books in the monorepo).

Related books

Book	Overlap
Networking	Deeper TCP/IP, routing, and lab topologies beyond host CLI
VCS	Git workflows used on the servers you administer
NixOS	Declarative hosts after you know classic Linux ops
Go / C	Systems programming that assumes this CLI fluency

How to use

1. Expand **Commands** → open a topic → open a command page.
2. Each Commands topic starts with an **Intro** page: overview, command table, and suggested starting points — then one page per command.
3. Expand **Editors** → pick an editor or workspace tool (vi, Neovim, tmux, Herdr, ...).
4. Expand **Scripting** → **Bash** for shell programming and automation beyond one-liners.
5. **Search** for a command or editor name when you already know what you need.
6. Prefer **worked examples** over option dumps — copy, adapt, verify on a server or VM.

Commands page template

Most command pages follow this shape:

1. Overview
2. Syntax
3. Common options
4. Safety (when destructive or privileged)
5. Key use cases (Core pages)
6. Examples with **named** explanations
7. Understanding output (when useful)
8. Notes & pitfalls

9. Related commands

10. Further reading (optional)

Short utilities may use a lighter form. See `content/_planning/command-page-template.md` for author depth tiers (Core / Standard / Light).

Commands — parts at a glance

Section	Focus
Help and docs	<code>man</code> , <code>info</code> , <code>help</code> , <code>apropos</code> , <code>whatis</code>
Files and paths	Navigation, copy/move, permissions, ACLs, attributes, search
Archives and compression	<code>tar</code> , <code>gzip</code> , <code>xz</code> , <code>zstd</code> , <code>zip</code> , <code>zcat</code> / <code>zgrep</code>
System information	Host identity, memory, sessions, hardware summaries
Processes and jobs	<code>ps</code> , <code>top</code> , signals, job control, priority
System monitoring	<code>iostat</code> , <code>vmstat</code> , <code>sar</code> , <code>strace</code> , <code>ltrace</code> , <code>perf</code>
Users and groups	Accounts, passwords, aging, <code>su</code> / <code>sudo</code>
Networking	<code>ip</code> , <code>ss</code> , DNS, SSH, transfer, firewalls, capture, <code>ethtool</code>
Scheduling	<code>cron</code> , <code>crontab</code> , <code>at</code> , <code>anacron</code>
Logging	<code>logger</code> , <code>logrotate</code> (see also <code>journalctl</code>)
Hardware	<code>lshw</code> , <code>lspci</code> , <code>dmidecode</code> , <code>hdparm</code>
Storage and filesystems	<code>lsblk</code> , LVM, LUKS, <code>parted</code> , SMART, <code>grow/tune ext*</code>
Terminals and multiplexers	<code>tmux</code> , <code>screen</code> , and related terminal tools
Text and pipes	<code>grep</code> , <code>sed</code> , <code>awk</code> , <code>jq</code> , <code>pv</code> , <code>sort/uniq</code> pipelines
Packages	<code>apt/dpkg</code> , <code>dnf/rpm</code> , <code>yum</code> , server-oriented <code>snap</code>
Services and runtime	<code>systemctl</code> , <code>journalctl</code> , timers, modules, <code>podman</code>
Shell commands	<code>echo/printf</code> , history, <code>type</code> , <code>export</code> , <code>time</code> , directory stack
Security	SELinux mode/labels, file capabilities (RHEL-family focused)
Appendices	Cheatsheets and longer extras

Editors (this book)

Terminal-native editing — classic (`vi`/`vim`/`nano`), modern modal (Neovim, LazyVim, Helix, Kakoune), GUI-speed **Zed**, plus workspace tools (**tmux**, **Herd**). Start under **Editors** → **Intro** for paths (sysadmin emergency, Vim family, AI agents).

Scripting → Bash (this book)

Shell *programming* beyond one-liners: shells vs terminals, first scripts, variables, control flow, functions, and a workbook. Use **Commands** for individual utilities; use **Bash** when you are writing scripts.

Conventions

- Examples assume a modern GNU/Linux **server** userland (Ubuntu 22.04/24.04 Server compatible unless noted).
- Prefer **ip** / **ss** / **systemctl** over legacy **ifconfig** / **netstat** / SysV-only workflows.
- Ubuntu-first package names where they matter; note Fedora/RHEL variants when tools differ.
- Worked examples over option dumps.
- Safety notes on destructive or privileged operations.

Formats

HTML, PDF, and EPUB when built via the monorepo portal.

Commands

Help and Docs

Intro

Intro

This part covers how to find answers **on the system itself** before searching the web. Offline docs match the versions you actually installed — critical when flags change between releases.

Commands in this part

Command	Role
<code>man</code>	<code>man</code> displays the system manual.
<code>help</code> (shell builtin)	<code>help</code> shows documentation for shell builtins — commands implemented inside the shell itself (<code>cd</code> , <code>export</code> , <code>read</code> , <code>[[</code> , ...
<code>info</code>	<code>info</code> reads documentation in Info (Texinfo) format.
<code>apropos</code>	<code>apropos</code> searches man page names and short descriptions for keywords.
<code>whatis</code>	<code>whatis</code> prints single-line descriptions from the man-db <code>whatis</code> index.

Suggested starting points

1. Quick hint: `whatis cmd` or `cmd --help`.
2. Full story: `man cmd` (try section 5 for configs, 8 for admin tools).
3. Forgot the name: `apropos keyword`, then open the best hit with `man`.
4. Shell builtin: `type cmd`, then `help cmd` if it is a builtin.
5. GNU deep dive: `info` for `coreutils`, `bash`, and related tools.

Related parts

- `type / command -v` (Shell commands) — resolve names
- Book introduction — how pages are structured

Continue with the individual command pages in this part.

man

Overview

`man` displays the system manual. Pages are organized into **sections** covering user commands, system calls, libraries, file formats, and admin tools. It remains the authoritative offline reference on Linux — faster and more version-accurate than random web snippets for the packages you actually have installed.

Most pages are viewed through a pager (usually `less`). Press `q` to quit, `/` to search.

Syntax

```
man [section] name
man [options] name
man -k keyword      # same as apropos
man -f name          # same as whatis
```

Sections (common)

Sec	Content	Example
1	User commands	<code>man 1 printf</code>
2	System calls	<code>man 2 open</code>
3	Library functions	<code>man 3 printf</code>
4	Special files / devices	<code>man 4 null</code>
5	File formats / configs	<code>man 5 passwd</code>
6	Games	—
7	Overviews / conventions	<code>man 7 signal</code>
8	Admin commands	<code>man 8 mount</code>
9	Kernel routines (some systems)	—

When the same name exists in multiple sections (e.g. `printf` in 1 and 3, `passwd` in 1 and 5), specify the section number.

Common Options

Option	Description
<code>-k / --apropos</code>	Search short descriptions
<code>-f / --whatis</code>	One-line summary for a known name
<code>-a</code>	Show all matching sections in turn
<code>-w / --path</code>	Print path to the man file, don't open it
<code>-W</code>	List all matching paths
<code>-K</code>	Full-text search of page bodies (slow)
<code>-P pager</code>	Override pager (<code>less</code> , <code>cat</code> , ...)
<code>-L locale</code>	Force locale for translated pages
<code>MANPAGER / PAGER</code>	Environment override for pager
<code>MANWIDTH</code>	Force formatting width (useful in scripts)

Examples with Explanations

Open a page

```
man ls
man 5 sshd_config
man 8 systemctl
man man
```

`sshd_config` lives in section 5 (file formats), not as a user command.

Disambiguate sections

```
man -a printf      # cycle through every section that has printf
man 3 printf        # C library specifically
whatis printf       # see which sections exist
man -w -a passwd    # paths for all passwd pages
```

Search when you forget the name

```
man -k resize
man -k 'disk usage'
apropos network interface
man -K 'AddressFamily' # full-text; can take a while
```

Locate the source file

```
man -w ls
# e.g. /usr/share/man/man1/ls.1.gz
zcat "$(man -w ls)" | head
```

Useful when debugging missing or outdated pages after package installs.

Readable width and plain text dump

```
MANWIDTH=80 man ls
man ls | col -b > /tmp/ls.txt    # strip backspaces/overstrike
man -P cat ls | head -40
```

HTML / browser (if supported)

```
man -H ls                      # uses browser from $BROWSER or similar
man -Tutf8 ls | less -R
```

Scripting existence checks

```
if man -w journalctl >/dev/null 2>&1; then
    echo "journalctl man page present"
fi
```

Pager keys (less)

Key	Action
q	Quit
/pattern	Search forward
?pattern	Search backward
n / N	Next / previous match
g / G	Top / bottom
Space / b	Page down / up
h	Help

Notes / Pitfalls

- Empty or stale results after installing packages: rebuild the index with `sudo mandb` (or `sudo makewhatis` on some BSDs / older systems).
- Container images often ship **without** man pages to save space. Install `man-db` and the relevant `*-doc` / man packages, or read docs on the host.
- Section order and default “first match” can surprise you — prefer explicit `man 5 name` for configs.
- Locale: if a translated page is incomplete, force English with `LANG=C man ...` or `man -L C ...`.
- `man` is not for shell builtins (`cd`, `[[`, `declare`) — use `help` (bash) instead.

2026-relevant notes

- `Systemd`, `iproute2`, and `Podman` pages are high quality; prefer `man systemctl`, `man ip-link`, `man podman-run` over outdated blog posts for flag accuracy.
- Many modern tools also ship `--help` that is good enough for flags; use `man` when you need semantics, exit codes, and file formats.
- Container and snap sandboxes may not see host man pages; read docs in the environment where the tool is installed.

Related Commands

- `info` — Texinfo hypertext manuals (GNU tools)
- `help` — shell builtins
- `apropos` / `whatis` — search and one-liners
- `tldr` / `cheat` — community example sheets (if installed)
- `mandb` — rebuild the `whatis` database

Additional Resources

- `man man`, `man man-pages`, `man 7 man-pages`
- [Linux man-pages project](#)

help (shell builtin)

Overview

`help` shows documentation for **shell builtins** — commands implemented inside the shell itself (`cd`, `export`, `read`, `[[`, `declare`, job control, etc.), not as separate binaries under `/usr/bin`. External programs use `--help` or `man` instead.

If `man cd` looks wrong or minimal, you are probably looking at a stub; the real reference is `help cd` (bash) or the shell's own manual.

Syntax

```
help [-dms] [pattern ...]
```

Bash-oriented options (check `help help` on your shell):

Option	Description
<code>-d</code>	Short description only
<code>-m</code>	Manpage-like layout
<code>-s</code>	Short usage synopsis
<i>pattern</i>	Builtin name or pattern to match

Common Options / discovery helpers

Command	Description
<code>help</code>	List topics / usage overview
<code>help name</code>	Full builtin help
<code>type name</code>	How the shell resolves name
<code>type -a name</code>	All candidates (builtin + PATH)
<code>command -v name</code>	Path or builtin indicator
<code>compgen -b</code>	List bash builtins
<code>enable -a</code>	Enabled/disabled builtins

Examples with Explanations

Read builtin docs

```
help cd
help [[
help declare
help read
help -m printf      # man-like layout for the builtin printf
help | less         # browse all topics
```

Builtin vs external

```
type cd              # builtin
type ls              # usually /usr/bin/ls (hashed)
type -a echo         # may show builtin and /usr/bin/echo
type -a kill         # often both: shell kill + /bin/kill
help cd              # works for builtin
ls --help            # external GNU flag
man ls               # full external manual
```

Prefer `type / command -v` in scripts over `which` for resolving names.

List builtins (bash)

```
compgen -b | column
compgen -b | grep -E '^(cd|pushd|popd|dirs)$'
enable -a | head
```

Disable / re-enable a builtin (rare, debugging)

```
enable -n echo       # force external echo next
type echo
enable echo           # restore builtin
```

Useful when testing whether a weird `echo` behavior comes from the builtin or a shadowed binary.

Conditional help in interactive setups

```
# only define an alias if the real command exists
if type -P rg >/dev/null; then
    alias grep='rg'
fi
```

`type -P` searches PATH only (ignores builtins/aliases/functions).

Job-control and shell language topics

```
help jobs
help fg
help bg
help wait
help trap
help set
help shopt
```

These are not separate man pages in the same way; learn them via `help + man bash`.

Notes / Pitfalls

- `help` is primarily for **interactive** humans. Non-interactive scripts still *run* builtins; they just don't need `help`.
- `zsh` uses `run-help` (often bound to `Esc-h`) and different docs; `fish` integrates web/local docs under `help`.
- `help printf` is the **shell** `printf`; `man 1 printf` / `man 3 printf` are different tools/APIs.
- Aliases and functions shadow names: `type foo` before assuming you are reading the right `help`.
- BusyBox / dash environments have smaller builtin sets; don't assume full bash `help` text.

2026-relevant notes

- Bash 5.x remains the default interactive shell on many servers; `help` text tracks your installed bash, not a website.
- For deep language reference (arrays, coprocs, `mapfile`, loadable builtins), combine `help` with `man bash` / `info bash`.
- In containers, shells are often `bash` or `sh` → `dash`; verify with `echo $0` and `help` availability.

Related Commands

- `man bash` / `info bash` — full shell language reference
- `type` / `command -V` — resolve names
- `whatis` / `man` — external commands
- `compgen` / `complete` — completion and name generation
- `set` / `shopt` — shell options (see `help set`)

Additional Resources

- `help help`, `man bash`
- [Bash Reference Manual](#)

info

Overview

`info` reads documentation in **Info** (Texinfo) format. GNU projects often put their deepest manuals here — structured as a tree of **nodes** with menus and cross-references — while `man` pages stay as flatter reference sheets.

For `coreutils`, `bash`, and many GNU tools, `info` can be more complete than the `man` page. Navigation is keyboard-driven and different from `less`; invest a few minutes in the keys once.

Syntax

```
info [options] [menu-item ...]
info [options] -f file [node]
info [options] --index-search=string
```

Common Options

Option	Description
<code>-f file</code>	Open a specific Info file
<code>-n node</code>	Start at named node
<code>-a</code>	Use all matching manuals
<code>-k string / --apropos</code>	Search indices for string
<code>-w</code>	Show location of Info file
<code>--index-search=str</code>	Go to index entry
<code>--show-options</code>	Jump to command-line options node (when present)
<code>--subnodes</code>	Dump node and subnodes (non-interactive)
<code>-o file</code>	Write output to file
<code>--vi-keys</code>	Vi-like key bindings

Navigation Keys (standalone info)

Key	Action
?	List commands
h	Tutorial
q	Quit
n / p	Next / previous node at this level
u	Up to parent node
l	Last node (history back)
] / [	Forward / back in document order
m	Menu item by name
f	Follow cross-reference
s / /	Search
Space / Del	Scroll
d	Directory node (top-level catalogue)

Emacs users often prefer `M-x info` inside Emacs; keys differ slightly.

Examples with Explanations

Open a manual

```
info ls
info coreutils
info bash
info libc
```

Tab completion or the directory node (`info` with no args, then browse) helps discovery.

Jump to options / a node

```
info --show-options ls
info -n 'Invoking ls' coreutils
info -f coreutils -n 'Directory listing'
```

Search

```
info --apropos=printf
info -k symlink
info --index-search=regex
```

Where is the file?

```
info -w coreutils
# e.g. /usr/share/info/coreutils.info.gz
```

Non-interactive extract

```
info --subnodes -o /tmp/ls-info.txt ls
info -o - --show-options ls | head -50
```

Useful for grepping offline or pasting into notes.

Prefer man for quick flags, info for depth

```
man ls          # quick reference
info ls         # full GNU narrative + examples
info coreutils  # whole suite manual
```

Understanding the Info tree

1. **Directory node** — catalogue of installed manuals (d).
2. **Top node** — start of one manual.
3. **Menus** — child nodes (chapters/sections).
4. **Cross-references** — `*note ...` links between topics.
5. **Indices** — topic/command/option indices for search.

Think of it as a hypertext book, not a single scrollable man page.

Notes / Pitfalls

- Missing docs: install packages like `info`, `bash-doc`, `coreutils` info files (`*-doc` on some distros).
- People open `info` once, hate the UI, and never return — spend five minutes on `h / ?`.
- Not every tool has an Info manual; many only ship man pages.
- `pinfo` is a friendlier alternative viewer if you install it.
- Dumping with `--subnodes` can be huge (`libc`); narrow to a node first.

2026-relevant notes

- Man pages improved a lot for `systemd/iproute2`; Info still wins for **GNU coreutils**, **findutils**, **bash**, **gdb**, **gcc** deep dives.
- In CI/containers, Info files are often omitted; generate or fetch docs on a full workstation.
- Some projects moved primary docs to HTML/web; Info remains the classic offline GNU format.

Related Commands

- `man` — traditional manual pages
- `pinfo` — alternative Info browser
- `apropos` / `whatis` — man-db search / one-liners
- `help` — shell builtins
- `info --help` — viewer options

Additional Resources

- `info info` — the Info reader manual itself
- [GNU Texinfo / Info documentation](#)

apropos

Overview

`apropos` searches man page **names and short descriptions** for keywords. Use it when you remember *what you want to do* but not the command name. It is equivalent to `man -k`.

It queries the **whatis database** built by `mandb`. If that DB is missing or stale, results will be empty or outdated — rebuild it rather than assuming the command does not exist.

Syntax

```
apropos [options] keyword...
man -k [options] keyword...
```

Common Options

Option	Description
<code>-a</code>	Require every keyword to match (AND)
<code>-r</code>	Treat keywords as regular expressions
<code>-w</code>	Shell-style wildcards
<code>-s list</code>	Limit to sections (e.g. 1,8)
<code>-e</code>	Exact match on page name/description token
<code>-l</code>	Do not truncate long descriptions
<code>-C file</code>	Alternate man config file

Exact flags can vary slightly by man-db version; check `man apropos` on your host.

Examples with Explanations

Basic keyword search

```
apropos password
apropos socket
apropos 'disk usage'
man -k resize
```

Quotes help when the phrase contains spaces or shell metacharacters.

AND multiple terms

```
apropos -a network interface
apropos -a list directory
```

Without `-a`, multiple words are often OR'd (implementation-dependent noise). Prefer `-a` to narrow results.

Limit by section

```
apropos -s 8 mount           # admin tools about mount
apropos -s 1,8 kill          # user + admin kill-related pages
apropos -s 5 passwd          # file format, not the passwd(1) command
```

Regex and wildcards

```
apropos -r 'zip$'            # names/descriptions ending in zip
apropos -r '^ip\b'
apropos -w 'sched*'
```

Reduce noise to section-1 commands

```
apropos copy | grep -E '^S\s+(1\)' | head -20
apropos -s 1 copy | head
```


Rebuild empty database

```
sudo mandb
# older / alternate systems:
# sudo makewhatis
apropos printf
whatis ls
```

Containers and minimal images often need `man-db` installed **and** `mandb` run once.

Pipeline into selection

```
cmd=$(apropos -s 1 disk | fzf | awk '{print $1}')
man "$cmd"
```

Handy on interactive workstations with `fzf`.

Compare with full-text search

```
apropos AddressFamily      # only names + short desc
man -K AddressFamily       # body text; much slower
```

Use `-K` when you know a config keyword but not which man page owns it.

Notes / Pitfalls

- “nothing appropriate” usually means **stale/missing mandb**, not that Linux lacks the tool.
- Descriptions are short; false positives are common (**t**ime matches many pages). Narrow with `-s` and `-a`.
- Localized `whatis` DBs may hide English-only phrasing; try `LANG=C apropos ....`
- `apropos` does **not** search shell builtins; use `help` / `compgen -b` for those.
- Package docs not installed → no man page → not in the index.

2026-relevant notes

- Prefer modern names in queries: `apropos systemd`, `apropos nft`, `apropos iproute` rather than only legacy `ifconfig/iptables` wording if you want current tooling.
- On immutable / ostree systems, man pages may live in different layers; ensure the layer with docs is available before blaming `apropos`.
- AI chat is fine for discovery, but `apropos` + `man` match **your installed version’s** flags.

Related Commands

- `whatis / man -f` — one-line summary for a known name
- `man -K` — full-text search of page bodies
- `man` — open the full page
- `info --apropos` — search Texinfo manuals
- `mandb` — rebuild the index

Additional Resources

- `man apropos`, `man man`, `man mandb`

whatis

Overview

`whatis` prints **single-line descriptions** from the man-db `whatis` index. Use it when you want a one-sentence reminder of what a command or page is without opening the full manual. Same idea as `man -f`.

It answers “what is this name?” — not “what command does X?” (that’s `apropos` / `man -k`).

Syntax

```
whatis [options] name...
man -f name...
```

Common Options

Option	Description
<code>-s list</code>	Only these sections (e.g. 1,8 or 5)
<code>-l</code>	Long / multi-line output when available
<code>-r</code>	Interpret name as regex (implementation-dependent)
<code>-w</code>	Wildcard match on names
<code>-v</code>	Verbose diagnostics
<code>-d</code>	Debug (man-db)

Examples with Explanations

Quick one-liners

```
whatis ls
whatis passwd
whatis passwd chmod mount
man -f ls # equivalent
```

Disambiguate by section

```
whatis passwd
whatis -s 1 passwd      # the command
whatis -s 5 passwd      # /etc/passwd format
whatis -s 2,3 open      # syscall / libc
```

Multiple names in scripts

```
for c in curl wget aria2c; do
    whatis "$c" 2>/dev/null || echo "$c: no man summary"
done
```

Missing database

```
# typical message: whatis: nothing appropriate
sudo mandb
whatis ls
whatis systemctl
```

Minimal Docker images: install `man-db` and relevant man packages first.

Combine with type / path

```
type mount
command -v mount
whatis mount
man 8 mount
```

`type` shows *how the shell runs it*; `whatis` shows *what the man index claims it is*.

Wildcard / regex (when supported)

```
whatis -w 'system*'
whatis -r '^ip$'
```

Verify pages after package install

```
sudo apt install --reinstall manpages manpages-dev # Debian/Ubuntu example
sudo mandb
whatis open openat
```

Understanding Output

Typical line shape:

```
ls (1)           - list directory contents
passwd (1)       - change user password
passwd (5)       - the password file
```

Columns: **name**, (**section**), **short description**. Multiple lines appear when the name exists in several sections.

Exit status is non-zero when no entry matches (handy in scripts).

Notes / Pitfalls

- No output / “nothing appropriate” → run **sudo mandb**, install docs packages, or wrong name.
- Builtins like **cd** may have thin or missing **whatis** entries; use **help cd**.
- Descriptions can be outdated relative to a newer binary if packages diverged; open **man** when accuracy matters.
- **whatis** does not search descriptions for keywords — use **apropos**.
- Localized systems may show translated one-liners; use **LANG=C whatis ...** for English.

2026-relevant notes

- Still the fastest “sanity check” that a tool is installed *with documentation* on the box.
- Pair with **command -v** in onboarding scripts: binary present + **whatis** present = usable offline reference.
- Some distros split man pages into ***-doc** packages (especially libraries); **whatis** missing does not always mean the binary is missing.

Related Commands

- **apropos / man -k** — keyword search across descriptions
- **man** — full page
- **info** — Texinfo manuals
- **command -v / type** — how the shell resolves a name

- `mandb` — rebuild the `whatis` database

Additional Resources

- `man whatis`, `man man`

Files and Paths

Intro

Intro

Navigate the filesystem, create and move data, inspect metadata, search by name or content, and manage permissions (mode bits, ACLs, and attributes). This is the daily CLI surface for almost every task.

Commands in this part

Command	Role
<code>pwd</code>	<code>pwd</code> (print working directory) writes the absolute path of the current directory.
<code>cd</code>	<code>cd</code> (change directory) sets the shell's current working directory.
<code>ls</code>	<code>ls</code> lists directory contents.
<code>tree</code>	<code>tree</code> prints a directory hierarchy as an indented tree.
<code>mkdir</code>	<code>mkdir</code> (make directory) creates new directories.
<code>rmdir</code>	<code>rmdir</code> removes empty directories only.
<code>touch</code>	<code>touch</code> updates file timestamps and, by default, creates empty files that do not exist.
<code>cp</code>	<code>cp</code> copies files and directories.
<code>mv</code>	<code>mv</code> renames or moves files and directories.
<code>rm</code>	<code>rm</code> removes directory entries (unlinks files).
<code>cat</code>	<code>cat</code> (concatenate) reads files sequentially and writes them to standard output.
<code>less</code>	<code>less</code> is a pager: it displays text one screen at a time with searching, scrolling both ways, and optional follow mode.
<code>more</code>	<code>more</code> is a simple historical pager that displays text one screen at a time.
<code>chmod</code>	<code>chmod</code> changes file mode bits (permissions) and special bits (setuid, setgid, sticky).

Command	Role
<code>chown</code>	<code>chown</code> (change owner) sets the user and/or group ownership of files and directories.
<code>stat</code>	<code>stat</code> displays detailed inode metadata: size, ownership, mode, timestamps, device/inode numbers, links, and...
<code>file</code>	<code>file</code> classifies files by inspecting content magic (and optionally the filesystem type), not just the name extension.
<code>find</code>	<code>find</code> walks a directory tree and selects files by name, type, time, size, permissions, and more.
<code>fd</code>	<code>fd</code> is a fast, user-friendly alternative to <code>find</code> .
<code>locate</code>	<code>locate</code> finds files by name using a prebuilt database (historically <code>mlocate</code> ; on modern Ubuntu often <code>plocate</code>).
<code>ripgrep (rg)</code>	<code>ripgrep (rg)</code> recursively searches the filesystem for a regex pattern.
<code>ln</code>	<code>ln</code> creates links: hard links (default) or symbolic links (<code>-s</code>).
<code>readlink</code>	<code>readlink</code> prints the target of a symbolic link, or with GNU flags like <code>-f/-e/-m</code> , canonicalizes paths by resolving...
<code>realpath</code>	<code>realpath</code> prints the resolved absolute path of each argument, expanding <code>.</code>
<code>basename</code>	<code>basename</code> strips the directory portion (and optionally a suffix) from a path, leaving the final component.
<code>dirname</code>	<code>dirname</code> removes the last path component, returning the directory portion of a path.
<code>which</code>	<code>which</code> locates executables by searching the <code>PATH</code> environment variable and printing the path of the first match.
<code>alias / unalias</code>	<code>alias</code> is a shell builtin that creates command shortcuts in the current shell session.
<code>install</code>	<code>install</code> copies files with explicit mode (and optional owner/group) and can create destination directories.
<code>df</code>	<code>df</code> reports filesystem disk space usage for mounted filesystems.

Command	Role
<code>du</code>	<code>du</code> estimates file and directory space usage by walking the tree.
<code>bat</code>	<code>bat</code> is a modern <code>cat</code> replacement with syntax highlighting, git integration, line numbers, and automatic paging.
<code>eza</code>	<code>eza</code> is a modern replacement for <code>ls</code> (community successor to <code>exa</code>) with colors, git status, icons (optional), tree...
<code>sha256sum</code>	<code>sha256sum</code> computes or verifies SHA-256 message digests.
<code>base64</code>	<code>base64</code> encodes binary data to ASCII text, or decodes it back.
<code>cmp</code> <code>getfacl</code> / <code>setfacl</code>	<code>cmp</code> compares two files byte by byte. POSIX ACLs extend the classic owner/group/other mode bits with named user and group entries.
<code>chattr</code> / <code>lsattr</code>	<code>chattr</code> sets ext file attributes (immutable, append-only, no-dump, etc.); <code>lsattr</code> lists them.
<code>xxd</code>	<code>xxd</code> creates a hex dump of a file (or stdin) and can reverse a dump back to binary.
<code>strings</code>	<code>strings</code> extracts printable character sequences from binary files.
<code>namei</code>	<code>namei</code> walks a path component by component, showing whether each element is a file, directory, or symlink — and...
<code>umask</code>	<code>umask</code> sets the file mode creation mask for the current shell: bits that are turned <i>off</i> when new files and...

Suggested starting points

1. Orientation: `pwd`, `ls`, `cd`, `tree` / `eza`.
2. Mutate carefully: `cp`, `mv`, `rm`, `mkdir` (prefer dry-run / backups when unsure).
3. Permissions: `chmod`, `chown`, then ACLs (`getfacl`) or attributes (`chattr`/`lsattr`) when modes are not enough.
4. Find things: `find` or `fd` for live trees; `locate` for indexed names; `rg` for content.
5. Space: `df`, `du` (and later Storage tools like `ncdu`).

Related parts

- Storage and filesystems — block devices, mounts, LVM
- Text and pipes — process file contents
- Security — SELinux labels when DAC is not the issue

Continue with the individual command pages in this part.

pwd

Overview

`pwd` (print working directory) writes the absolute path of the current directory. Shells often implement `pwd` as a **builtin** that can show a **logical** path (may include symlink components via `$PWD`) or a **physical** path (symlinks resolved).

Use it after deep navigation, in scripts that log context, and whenever logical vs physical paths matter (bind mounts, symlinked project roots).

Syntax

```
pwd [options]
```

Common Options

Option	Description
<code>-L</code>	Logical path (<code>\$PWD</code> ; may retain symlink names) — often default for the builtin
<code>-P</code>	Physical path (resolve all symlinks)
<code>--help</code>	Help (external <code>/bin/pwd</code>)

`help pwd` documents the **bash builtin**; `man pwd` documents the external binary. Behavior can differ slightly.

Key Use Cases

1. Confirm location after many `cd`s
2. Capture cwd in scripts and logs
3. Compare logical vs physical paths through symlink trees
4. Build absolute paths for configs and tooling
5. Debug “wrong directory” issues in containers and chroots

Examples with Explanations

Basics

```
pwd
pwd -P
pwd -L
type pwd          # builtin or external?
/bin/pwd -P      # force external binary
```

Capture in scripts

```
current_dir=$(pwd)
current_dir=$PWD          # logical; no subprocess
phys=$(pwd -P)
echo "running in $phys"
```

Prefer "\$PWD" when you want the logical path without spawning a process; call `pwd -P` when you need the real path.

Symmlink illustration

```
mkdir -p /tmp/realdir
ln -sfn /tmp/realdir /tmp/linkdir
cd /tmp/linkdir
pwd -L    # often /tmp/linkdir
pwd -P    # /tmp/realdir
ls -ld .  # may still show link path depending on tools
```

Prompt and logging recipes

```
# Show physical path in a custom prompt experiment
PS1='\u@\h $(pwd -P)$ '

# Log absolute cwd with a command
echo "$(date -Is) cwd=$(pwd -P) cmd=$*" >> ~/cmd.log
```

Resolve then act

```
# Ensure absolute before relative ops
cd "$(pwd)/subdir"

# Physical home resolution
cd -P ~
pwd -P

# Fail if cwd vanished (deleted out from under the shell)
pwd -P || echo "cwd invalid"
```

Compare with realpath

```
pwd -P
realpath .
realpath -e .
```

`realpath .` is often equivalent to physical `pwd`; `realpath` also works on arbitrary paths, not just `pwd`.

Directory stack awareness

```
pushd /var/log >/dev/null
pwd
popd >/dev/null
pwd
dirs -v
```

Notes / Pitfalls

- Builtin vs `/bin/pwd` can disagree on default `-L/-P` semantics; check `type pwd` and `help pwd`.
- If the directory was deleted while you were inside it, `pwd` may error or show a stale `$PWD`.
- Don't mix logical and physical paths blindly in scripts (e.g. `cd` by logical path, then `pwd -P` for logs).
- `cd` following symlinks leaves a logical path unless you use `cd -P` or `set -o physical` (bash).
- In subshells, `pwd` changes don't affect the parent: `(cd /tmp && pwd)` vs later `pwd`.

2026-relevant notes

- Containers: `WORKDIR` in the image defines initial `pwd`; `pwd` inside the container reflects that namespace's mounts.

- Network filesystems and bind mounts: physical paths help correlate with host mounts (`findmnt`, `df`).
- Pair with `realpath` for **file** paths; reserve `pwd` for **directory** context.

Related Commands

- `cd` — change directory
- `realpath` — resolve arbitrary paths
- `readlink -f` — canonical path (GNU)
- `dirname` / `basename` — split path components
- `ls` — list contents
- `findmnt` — what is mounted where

Additional Resources

- `help pwd` (bash), `man pwd`

cd

Overview

`cd` (change directory) sets the shell's current working directory. It is a **shell builtin**, not an external program — each shell (`bash`, `zsh`, `dash`) implements it. Directory changes never affect other already-running processes; only the current shell (and its children) see the new `cwd`.

Syntax

```
cd [-L|[-P [-e]] [-@]] [directory]
cd -
```

Common Options / operands

Option / form	Description
<code>-L</code>	Follow logical path (default); retain symlink components
<code>-P</code>	Physical structure; resolve symlinks when determining path
<code>-e</code>	With <code>-P</code> , exit non-zero if physical path cannot be determined
<code>-</code>	Switch to previous directory (<code>\$OLDPWD</code>)
<code>~</code> or empty	Home directory (<code>\$HOME</code>)
<code>~user</code>	Home of <code>user</code>
<code>..</code> / <code>.</code>	Parent / current
<code>--</code>	End of options (path starts with <code>-</code>)

Key Use Cases

1. Navigate the filesystem in interactive shells
2. Anchor scripts to a known directory
3. Toggle between two locations with `cd -`
4. Control symlink resolution (`-L` vs `-P`)
5. Combine with `pushd`/`popd` for a directory stack

Examples with Explanations

Basic navigation

```
cd /usr/local/bin
cd /var/log
cd ~
cd                      # also home
cd ..
cd ../../
```

Previous directory

```
cd /etc
cd /var/log
cd -                      # back to /etc
cd -                      # toggle again
```

`$OLDPWD` holds the previous directory; `cd -` prints the new path in many shells.

Symmlink behavior

```
mkdir -p /tmp/real/sub
ln -sfn /tmp/real /tmp/link
cd /tmp/link/sub
pwd -L                      # /tmp/link/sub
pwd -P                      # /tmp/real/sub
cd -P /tmp/link/sub
pwd                          # physical
```

Paths that start with a dash

```
mkdir -- -weird
cd -- -weird
```

Script: run relative to script location

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$(readlink -f "$0")")"
pwd
```

```
# now relative paths are stable regardless of caller cwd
```

Script: require a directory

```
cd /srv/app || { echo "missing /srv/app" >&2; exit 1; }
```

pushd / popd alternative workflow

```
pushd /var/log
pushd /etc/systemd/system
dirs -v
popd
popd
```

cd alone does not maintain a stack; use pushd/popd when you need one.

One-liners

```
cd /usr/local/bin && pwd
cd ~alice
cd "$(dirname "$(realpath ./config.yml)")"
cd "$(mktemp -d)"          # disposable workspace
```

Understanding success and failure

- Success: no output (except cd - often prints the path).
- Common errors: permission denied, no such file, not a directory, deleted cwd.
- Exit status is non-zero on failure — always check in scripts (cd dir || exit 1).

Notes / Pitfalls

- cd in a subshell does not change the parent: (cd /tmp); pwd still shows the old path.
- Aliases like alias cd='cd -P' change defaults for interactive use only.
- CDPATH (if set) makes cd some search a list of base directories — surprising in scripts; unset or avoid relying on it in portable scripts.
- Broken symlinks and removed directories: physical cd -P fails more honestly.
- zsh auto-cd (typing a directory name) is a shell feature, not POSIX cd.

2026-relevant notes

- Containers and distroboxes: know whether you are on the host or in a named mount namespace before bulk relative operations.
- Network homes (`$HOME` on NFS) make frequent `cd` across mounts latency-sensitive; stay local when compiling.
- Prefer explicit absolute anchors in automation (`cd /opt/app`) over long chains of `cd ...`

Related Commands

- `pwd` — print working directory
- `pushd` / `popd` / `dirs` — directory stack
- `realpath` / `readlink` — resolve paths
- `ls` — list destination
- `findmnt` — mount context

Additional Resources

- `help cd` (bash)
- [Bash manual — Bourne Shell Builtins](#)

ls

Overview

`ls` lists directory contents. With long format, sorting, and classification options it is the primary interactive tool for inspecting the filesystem tree. Prefer **eza/lsd** if installed for richer defaults; every admin still needs classic `ls` for scripts and minimal systems.

Syntax

```
ls [options] [file...]
```

Common Options

Option	Description
<code>-l</code>	Long format (mode, links, owner, group, size, mtime, name)
<code>-a / -A</code>	All (include <code>./..</code>) / almost all (hide <code>./..</code> only)
<code>-h</code>	Human-readable sizes (with <code>-l</code>)
<code>-r</code>	Reverse sort
<code>-t / -S / -u / -c</code>	Sort by mtime / size / atime / ctime
<code>-R</code>	Recursive
<code>-1</code>	One entry per line
<code>-d</code>	List directories themselves, not their contents
<code>-F</code>	Classify (<code>/</code> dir, <code>*</code> exec, <code>@</code> symlink, ...)
<code>-i</code>	Inode numbers
<code>-s</code>	Size in blocks
<code>--color=auto</code>	Colorize when stdout is a tty
<code>-X</code>	Sort by extension
<code>-U</code>	Do not sort (directory order)
<code>--group-directories-first</code>	Dirs before files (GNU)
<code>-N</code>	Show raw names (less escaping)
<code>-L</code>	Follow symlinks for file info

Examples with Explanations

Everyday

```
ls
ls -la
ls -lah
ls -lt          # newest first
ls -lS          # largest first
ls -l
```

Hidden and directories

```
ls -A
ls -ld /var/log    # info about the dir itself
ls -ld .?.*        # hidden entries without . and ..
```

Sorting and time

```
ls -lt --time-style=long-iso
ls -ltu          # sort by atime
ls -lc           # sort by ctime
```

Recursive (careful)

```
ls -R /etc/ssh
# for deep trees prefer tree/find
tree -L 2
find /etc/ssh -maxdepth 2
```

Classification and inodes

```
ls -F
ls -li
ls -lH          # follow link for size/mtime display variants
```

Globs and multiple paths

```
ls -l *.md
ls -ld /etc /var /usr
ls /usr/bin/python*
```

Scripts: stable output

```
ls -l
ls -l --quoting-style=shell-escape
# parse carefully; prefer find -print0 for machine use
```

Color and aliases

```
ls --color=auto
alias ll='ls -lah --group-directories-first'
```

Compare modern alternatives

```
ls -lah
eza -lah --git          # if installed
```

Understanding long format

```
-rw-r--r-- 1 alice alice 1234 2026-01-15 10:02 file.txt
\ type+mode  links owner group size mtime      name
```

Leading type char: **-** file, **d** directory, **l** symlink, **b**/c devices, **p** fifo, **s** socket.

Notes / Pitfalls

- Output columns are for humans; parsing `ls` in scripts is fragile — use `find`, `stat`, or shell globs.
- Locale affects sort order and date formats; set `LC_ALL=C` for stable ASCII sort.
- `ls` of a symlink without `-l` may show the name only; details need `-l` / `-L`.
- Huge directories: `ls` sorts by default and can be slow; try `-U` or `find`.
- SELinux systems: `-Z` shows security context (GNU `ls`).

2026-relevant notes

- Many desktops alias `ls` to colorized forms; use `command ls` when you need vanilla behavior.
- `eza/lsd/ls++` improve UX; still teach classic flags for SSH to minimal servers.
- Combine with `bat/less -R` when piping colorized output.

Related Commands

- `eza / lsd` — modern listing
- `tree` — hierarchical view
- `find / fd` — programmable selection
- `stat` — precise metadata
- `dir / vdir` — alternate coreutils front-ends
- `namei` — path component walk

Additional Resources

- `man ls`

tree

Overview

`tree` prints a directory hierarchy as an indented tree. It is excellent for quick structural overviews, documentation screenshots, and seeing project layout without a full recursive `ls`. Often an optional package (`sudo apt install tree` / `dnf install tree`).

For huge trees, limit depth (`-L`) and prune noisy dirs (`-I`). For size-oriented exploration, prefer `du`, `dust`, `duf`, or `ncdu`.

Syntax

```
tree [options] [directory ...]
```

Common Options

Option	Description
<code>-L n</code>	Max depth
<code>-a</code>	Include hidden files
<code>-d</code>	Directories only
<code>-f</code>	Print full path prefix for each entry
<code>-i</code>	No indentation lines (plain list with nesting)
<code>-h</code>	Human-readable sizes (with size options)
<code>-s</code>	Raw sizes
<code>--du</code>	Report directory sizes (when supported)
<code>-P pattern</code>	Include only matching files
<code>-I pattern</code>	Exclude pattern (often <code> </code> -separated)
<code>-C / -n</code>	Color on / off
<code>-H url</code>	HTML output with base URL
<code>-J / -X</code>	JSON / XML (newer tree)
<code>--gitignore</code>	Honor <code>.gitignore</code> (newer tree)
<code>-p / -u / -g</code>	Permissions / user / group
<code>-D</code>	Show modified time
<code>--dirsfirst</code>	List directories before files

Examples with Explanations

Shallow overview

```
tree -L 2
tree -L 3 /etc
tree -d -L 3 /etc/systemd
```

Hide dependency / VCS noise

```
tree -I 'node_modules|.git|dist|target|.venv|__pycache__'
tree -L 2 -I 'node_modules|.git'
```

Hidden files and permissions

```
tree -a -L 1 ~
tree -pug -L 2 /var/www
```

Sizes

```
tree -h -L 2 --du          # if your tree supports --du
tree -h -L 1 /var
```

Full paths and filtering

```
tree -f -L 2
tree -P '*.service' -L 3 /etc/systemd
tree -P '*.md' -I node_modules
```

HTML report

```
tree -H '.' -L 2 -o /tmp/tree.html
# open /tmp/tree.html in a browser
```

JSON for tooling (when available)

```
tree -J -L 2 > tree.json
```

Compare with find / eza

```
tree -L 2
find . -maxdepth 2
eza -T -L 2           # modern ls alternative with tree mode
```

Document a project layout

```
tree -L 2 -I '.git|node_modules' > STRUCTURE.txt
```

Notes / Pitfalls

- Not installed by default on many minimal servers — package name is usually **tree**.
- Unbounded **tree** on / or large NFS shares is slow and noisy; always use **-L** / **-I**.
- Patterns for **-P/-I** are **not** full shell globs in all versions; check **man tree**.
- Color codes can break logs; use **-n** or redirect carefully.
- Do not treat **tree** as a security audit tool; it follows normal user permissions and may hide unreadable dirs.

2026-relevant notes

- Prefer **eza -T** or **lsd --tree** if you already use modern **ls** replacements; keep classic **tree** for portable docs.
- **--gitignore** support varies; upgrade **tree** or combine with **fd** for git-aware listing.
- In monorepos, generate shallow trees for READMEs rather than committing giant dumps.

Related Commands

- **ls -R** — recursive list
- **find** — programmable walk and filters
- **eza -T** / **lsd --tree** — modern tree views
- **du** / **dust** / **ncdu** — size-oriented
- **fd** — fast file find with ignores

Additional Resources

- man tree

mkdir

Overview

`mkdir` (make directory) creates new directories. With `-p` it creates parent paths as needed and does not error if the target already exists — the workhorse flag for scripts. Optional `-m` sets the mode for the created directory (subject to `umask` unless you understand the interaction).

Syntax

```
mkdir [options] directory...
```

Common Options

Option	Description
<code>-p, --parents</code>	Create parents; no error if existing
<code>-m MODE, --mode=MODE</code>	Set permission mode (e.g. 750, <code>u+rwX</code>)
<code>-v, --verbose</code>	Print a line for each created directory
<code>-Z</code>	Set SELinux context (when SELinux enabled)
<code>--context=CTX</code>	Set complete SELinux context

Key Use Cases

1. Create project and deploy directory trees
2. Idempotent path ensure in scripts (`-p`)
3. Create with restrictive permissions (`-m 700`)
4. Batch-create sibling dirs with braces
5. Scaffold log/cache layouts

Examples with Explanations

Basics

```
mkdir project
mkdir dir1 dir2 dir3
mkdir -v newdir
```

Parents

```
mkdir -p parent/child/grandchild
mkdir -p /var/lib/myapp/{data,cache,logs}
```

`mkdir -p` is safe to re-run: existing directories are left alone.

Permissions

```
mkdir -m 755 public_dir
mkdir -m 700 private_dir
mkdir -m 750 secure_dir
# mode applies to the leaf created; with -p, behavior for parents is version-specific -
# set explicitly if parents need special modes:
mkdir -p -m 755 /srv/app
chmod 750 /srv/app
```

Brace expansion scaffolds

```
mkdir -p logs/{app,nginx,db}
mkdir -p src/{cmd,internal,pkg} docs scripts
```

Script: ensure writable data dir

```
#!/usr/bin/env bash
set -euo pipefail
DATA_DIR=${DATA_DIR:-/var/lib/myapp}
mkdir -p "$DATA_DIR"
chmod 750 "$DATA_DIR"
```

SELinux (when relevant)

```
mkdir -Z /srv/webdata
# or restorecon after create on labeled systems
```

Parallel with install

```
install -d -m 755 /opt/myapp/bin
install -d -m 700 /opt/myapp/secrets
```

`install -d` is often preferred in packaging for explicit modes.

Failure cases

```
mkdir /proc/foo          # typically fails (pseudo-fs / perms)
mkdir /existing/file/sub  # fails if component is a file
```

Notes / Pitfalls

- Without `-p`, existing directory → error; missing parent → error.
- `umask` affects final mode when you don't fully specify bits; verify with `stat -c %a dir`.
- Race in concurrent scripts: two `mkdir` without `-p` can fail; prefer `-p` for idempotency.
- Creating under sticky dirs (`/tmp`) is fine; deleting others' dirs there is not.
- NFS root_squash: creating as root may result in `nobody` ownership.

2026-relevant notes

- In containers, create runtime dirs in entrypoints with `-p` rather than baking empty layers unless needed for ownership.
- `systemd RuntimeDirectory=` / `StateDirectory=` can replace hand-rolled `mkdir` in unit files — prefer unit directives for services.
- Immutable systems may only allow `mkdir` on writable mounts (`/var`, `/home`, `/tmp`).

Related Commands

- `rmdir` — remove empty directories
- `rm -r` — remove trees
- `install -d` — create dirs with mode/owner
- `chmod` / `chown` — adjust after create
- `mktemp -d` — secure temporary directories

- `ln -s` — link instead of nesting copies

Additional Resources

- `man mkdir`
- [GNU coreutils — mkdir](#)

rmmdir

Overview

`rmmdir` removes **empty** directories only. It is safer than `rm -r` when you want the kernel to refuse deletion if anything is still inside. For recursive removal of non-empty trees, use `rm -r` (carefully) or empty the tree first with `find/rm`.

Syntax

```
rmmdir [options] directory...
```

Common Options

Option	Description
<code>-p, --parents</code>	Remove directory and empty ancestors
<code>--ignore-fail-on-non-empty</code>	Do not error when a directory is non-empty
<code>-v, --verbose</code>	Print a line for each directory removed
<code>--help</code>	Help
<code>--version</code>	Version

There is **no** `-f` force that deletes contents — by design.

Key Use Cases

1. Remove empty directories safely
2. Tear down a path of empty parents (`-p`)
3. Verify a directory is empty (failure means “not empty” or other error)
4. Clean scaffolding left after build/install scripts
5. Safer automation than blanket `rm -rf`

Examples with Explanations

Basic removal

```
mkdir /tmp/empty_dir
rmdir /tmp/empty_dir
```

Non-empty fails

```
mkdir -p /tmp/d
echo x > /tmp/d/file
rmdir /tmp/d
# rmdir: failed to remove '/tmp/d': Directory not empty
rm /tmp/d/file
rmdir /tmp/d
```

Remove parent chain

```
mkdir -p /tmp/a/b/c
rmdir -p /tmp/a/b/c
# removes c, then b, then a if each is empty
```

Equivalent idea: `rmdir /tmp/a/b/c /tmp/a/b /tmp/a` from deepest up.

Verbose

```
rmdir -v empty1 empty2
```

Ignore non-empty (continue batch)

```
rmdir --ignore-fail-on-non-empty dir1 dir2 dir3
echo $?
```

Useful when cleaning many dirs and some still have files.

Test emptiness in scripts

```
if rmdir "$dir" 2>/dev/null; then
    echo "removed empty $dir"
else
    echo "not empty or missing: $dir"
fi
```

Combine with find (empty dirs only)

```
# list empty directories
find /tmp/project -type d -empty

# delete empty directories depth-first
find /tmp/project -depth -type d -empty -delete
# or
find /tmp/project -depth -type d -empty -exec rmdir {} +
```

Brace / multi targets

```
mkdir -p logs/{app,nginx,db}
# ... later, if empty:
rmdir logs/app logs/nginx logs/db logs 2>/dev/null
```

Notes / Pitfalls

- Hidden files (including `.gitkeep`, `.DS_Store`) make a directory non-empty.
- You need write permission on the **parent** directory to remove an entry.
- `rmdir -p` stops when it hits a non-empty ancestor; earlier empties are already gone.
- Never confuse with `rm -r`; `rmdir` will not delete files.
- Mount points and sticky directories (e.g. `/tmp`) follow special rules; you may not remove others' dirs there.

2026-relevant notes

- Build systems and containers often leave empty layers of dirs; `find -type d -empty` is the scalable cleanup.
- Prefer explicit empty-dir removal in deploy scripts over recursive force deletes.
- On immutable OS trees, empty-dir removal may be blocked by filesystem policy — expect `EROFS/EPERM`.

Related Commands

- `rm` — remove files and (with `-r`) trees
- `mkdir` — create directories
- `find -type d -empty` — locate empty dirs
- `rm -r` — recursive delete (dangerous)
- `unlink` — remove a single filesystem entry

Additional Resources

- `man rmdir`
- [GNU coreutils — rmdir](#)

touch

Overview

`touch` updates file timestamps and, by default, **creates** empty files that do not exist. Operators use it to create placeholders, nudge **make** rebuilds, normalize mtimes in tests, and clone timestamps from a reference file.

Linux files typically track at least **atime** (access), **mtime** (content modification), and **ctime** (metadata change). `touch` adjusts atime/mtime; ctime updates as a side effect of the metadata change.

Syntax

```
touch [options] file...
```

Common Options

Option	Description
<code>-a</code>	Change access time only
<code>-m</code>	Change modification time only
<code>-c, --no-create</code>	Do not create missing files
<code>-d STRING, --date=STRING</code>	Parse human-friendly time string
<code>-t [[CC]YY]MMDDhhmm[.ss]</code>	Set time in compact form
<code>-r REF, --reference=REF</code>	Copy timestamps from REF
<code>--time=WORD</code>	access/atime/use or modify/mtime
<code>-h</code>	Affect symlink itself (when supported)

Key Use Cases

1. Create empty files / placeholders
2. Update mtime to trigger build systems
3. Set explicit timestamps for tests and packages
4. Copy times from a reference file
5. Batch-create names (`touch file{1..5}`)

Examples with Explanations

Create or refresh

```
touch newfile
touch file1 file2 file3
touch file{1..5}
touch logs/.gitkeep
```

Do not create

```
touch -c missing_file      # no-op if absent; exit 0 typically
ls missing_file
```

atime / mtime only

```
touch -a file              # access time
touch -m file              # modification time
stat file
```

Date strings

```
touch -d '2020-01-01' file
touch -d '2 days ago' file
touch -d '2024-06-15 14:30:00' file
touch -d 'next Monday' file
```

GNU touch accepts many date-style strings.

Compact -t form

```
touch -t 202312201200 file
touch -t 202312201200.30 file
# [[CC]YY]MMDDhhmm[.ss]
```

Reference file

```
touch -r /etc/hosts mycopy
stat -c '%y %n' /etc/hosts mycopy
```

Make and empty targets

```
touch src/main.c          # force rebuild consumers of main.c
make
```

Scripts: ensure path exists as empty file

```
mkdir -p "$(dirname "$out")"
touch "$out"
```

Clear mtime into the past for cleanup tests

```
touch -d '30 days ago' /tmp/old.log
find /tmp -name 'old.log' -mtime +7
```

Understanding timestamps

```
stat file
stat -c 'mtime=%y atime=%x ctime=%z' file
```

Field	Meaning
atime	Last access (often lazy/relatime on modern mounts)
mtime	Last content modification
ctime	Last inode/metadata change (not “creation”)
birth	Creation time on some filesystems (<code>stat %w</code>)

relatime / **noatime** mount options mean `touch -a` may not behave as you expect for real reads; the explicit `touch -a` still sets atime.

Notes / Pitfalls

- Creating files requires write permission on the parent directory.
- `touch` on a directory updates the directory's timestamps, not children.
- Timestamp precision and timezone depend on filesystem and locale settings.
- NFS `root_squash` and permission quirks can make `touch` fail with `EACCES`.
- `touch` is not a substitute for writing content — use redirection or editors.

2026-relevant notes

- Build caches (Bazel, Nix, container layer caches) care deeply about mtime vs content hashes; know which model your toolchain uses.
- `touch -r` is handy when replaying reproducible package layouts.
- For pure “ensure file exists” in bash, `: >>file` or `umask + redirect` may be clearer than `touch` when you also write content.

Related Commands

- `stat` — display timestamps and inode data
- `date` — print/set system time; format strings
- `find -mtime/-mmin` — select by age
- `mkdir` — create directories
- `install` — install files with mode/owner
- `make` — rebuild based on mtimes

Additional Resources

- `man touch`
- [GNU coreutils — touch](#)

cp

Overview

cp copies files and directories. For large trees with incremental updates, prefer **rsync**. Mind directory semantics: trailing slashes and whether the destination already exists change results. Default is not recursive — use **-r/-R** or **-a** for trees.

Syntax

```
cp [options] source dest
cp [options] source... directory
cp [options] -t directory source...
```

Common Options

Option	Description
-a, --archive	Same as -dR --preserve=all (typical full tree copy)
-r, -R, --recursive	Copy directories recursively
-i, --interactive	Prompt before overwrite
-n, --no-clobber	Never overwrite
-u, --update	Copy only when source is newer or dest missing
-v, --verbose	Show files copied
-p	Preserve mode, ownership, timestamps (subset of archive)
--preserve=ATTR_LIST	Fine-grained preserve (mode,ownership,timestamps,links,xattr,context)
-L / -P / -H	Always follow / never follow / follow command-line symlinks
-l / -s	Hardlink / symlink instead of copying data
--reflink=auto	CoW lightweight clone when filesystem supports it
-T	Treat dest as a normal file (do not put source <i>into</i> dest dir)
-t DIR	Destination directory first
--parents	Create full parent path under dest

Safety

- `cp -a src dest` vs `cp -a src/ dest/` vs `cp -a src/. dest/` behave differently when `dest` exists.
- Dry-run mentally or use `rsync -n` for complex trees.
- Overwriting live configs can break services — copy to `.new` then `mv`.

Examples with Explanations

Files

```
cp file.txt file.bak
cp -i *.conf /etc/myapp/
cp -n important.dat /backup/      # do not overwrite
cp -u -v *.o build/
```

Directories

```
cp -a project/ project-backup/
cp -a src/. dest/                # copy *contents* into dest
cp -r --preserve=timestamps logs/ logs-copy/
```

Reflink (btrfs/xfs when available)

```
cp --reflink=auto big.img big-clone.img
```

Falls back to full copy if CoW clone is unsupported.

Preserve attributes selectively

```
cp --preserve=mode,timestamps src dest
cp -a --no-preserve=ownership src dest  # useful as non-root
```

Parents and target dir

```
cp --parents etc/ssh/sshd_config /backup/
# creates /backup/etc/ssh/sshd_config
cp -t /backup/ a.txt b.txt
```

Symlinks

```
cp -P link dest_link      # copy symlink itself
cp -L link dest_file      # copy the referent's data
```

One-liners

```
cp -a /var/www/html/. /var/www/html.bak/
cp -ai src/* dest/          # interactive archive-ish for files
install -D -m 644 app.conf /etc/app/app.conf  # alternative for install paths
```

Notes / Pitfalls

- Cross-filesystem copies rewrite data; hard links cannot span devices.
- Sparse files: GNU `cp` has `--sparse=always|auto|never` — important for VM images.
- Permissions: non-root cannot always preserve ownership; archive mode still copies content.
- Trailing slash myths: prefer explicit `src/.` into existing `dest/` when you mean “contents”.
- `cp` does not delete extraneous files in `dest` (mirroring needs `rsync --delete`).

2026-relevant notes

- For backups and sync, `rsync -aHAX --info=progress2` is usually better than raw `cp -a`.
- Reflink copies shine on btrfs/xfs with CoW; verify with `stat/filefrag` when testing.
- In containers, watch UID maps — preserved numeric owners may not match host users.

Related Commands

- `rsync` — incremental and remote-capable copy
- `install` — copy with explicit mode/owner
- `mv` — rename/move
- `scp` / `sftp` — remote copy over SSH
- `tar` — stream trees with ownership
- `dd` — block-level copy

Additional Resources

- `man cp`
- [GNU coreutils](#) — `cp`

mv

Overview

`mv` renames or moves files and directories. On the **same filesystem**, rename is typically atomic and instant (directory entry update). **Across filesystems**, `mv` copies then deletes, which is slower, non-atomic, and can change performance and link behavior.

Syntax

```
mv [options] source dest
mv [options] source... directory
mv [options] -t directory source...
```

Common Options

Option	Description
<code>-i, --interactive</code>	Prompt before overwrite
<code>-n, --no-clobber</code>	Do not overwrite
<code>-f, --force</code>	Do not prompt (default in many cases)
<code>-u, --update</code>	Move only when source is newer or dest missing
<code>-v, --verbose</code>	Print each move
<code>-b, --backup</code>	Backup existing destinations
<code>-S SUFFIX</code>	Backup suffix
<code>-t DIR</code>	Target directory first
<code>-T</code>	Treat dest as a normal file (not a directory)
<code>--strip-trailing-slashes</code>	Remove trailing slashes from sources

Key Use Cases

1. Rename files and directories
2. Move into another directory
3. Atomic replace of configs (`mv new current`)
4. Bulk relocate with shell globs
5. Safer overwrite policies (`-i` / `-n`)

Examples with Explanations

Rename and move

```
mv old.txt new.txt
mv file.txt /tmp/
mv chapter.md docs/
mv -t /backup/ *.log
```

Interactive / no-clobber

```
mv -i *.md docs/
mv -n src dest          # leave dest if it exists
```

Atomic replace pattern

```
cp app.conf app.conf.bak
install -m 644 app.conf.new /etc/myapp/app.conf.new
mv /etc/myapp/app.conf.new /etc/myapp/app.conf
```

Readers opening the path see old or new content, not a partial write (when same filesystem).

Verbose bulk

```
mv -v *.png ~/Pictures/
```

Backup on conflict

```
mv -b file.txt /dest/file.txt
# creates file.txt~ or numbered backups depending on options
```

Directory edge cases

```
mv dir existing_dir/    # moves dir *into* existing_dir
mv -T dir existing_file  # refuse treating dest as directory
```

Cross-device awareness

```
df -P file.txt /mnt/usb
mv file.txt /mnt/usb/      # copy+delete if different devices
```

Swap two files (bash)

```
mv a.tmp a.bak && mv a a.tmp && mv a.bak a
# or use a third temporary name carefully
```

Notes / Pitfalls

- Running processes that already hold an open file descriptor keep the **old inode** after you `mv` a replacement into place — they won't see new content until reopen.
- Cross-device moves break hard links and expand sparse files unless tools special-case them.
- Globs that accidentally match the destination directory can error or nest oddly — quote and order carefully.
- `mv` does not update application-internal paths stored inside files.
- Permissions: need write access on source's parent and dest's parent.

2026-relevant notes

- Deploy pattern: write `foo.new` → `fsync` → `mv foo.new foo` remains the standard atomic publish on POSIX local filesystems.
- For directory trees across mounts, prefer `rsync` then delete, or `tar` pipelines, for progress and resumability.
- On overlay/container layers, “rename atomicity” still holds within a layer but publishing images is a different model.

Related Commands

- `cp` — copy without removing source
- `rm` — delete
- `rename` — bulk rename (Perl and util-linux variants differ!)
- `rsync --remove-source-files` — move-like with `rsync` features
- `install` — install to final path with mode
- `ln` — links instead of moving

Additional Resources

- `man mv`

rm

Overview

`rm` removes directory entries (unlinks files). By default it does **not** remove directories unless `-r/-R` is given. There is no undelete in coreutils — recovery requires backups, snapshots, or filesystem forensics.

Treat `rm -rf` as a loaded tool: verify the path, prefer dry-runs, and avoid root experiments on broad globs.

Syntax

```
rm [options] file...
```

Common Options

Option	Description
<code>-r, -R, --recursive</code>	Remove directories and their contents
<code>-f, --force</code>	Ignore missing files; never prompt
<code>-i</code>	Prompt before every removal
<code>-I</code>	Prompt once before removing many / recursive
<code>-v, --verbose</code>	Print each path as it is removed
<code>-d</code>	Remove empty directories (like <code>rmdir</code>)
<code>--one-file-system</code>	Do not recurse into different mount points
<code>--preserve-root</code>	Refuse to recursively remove / (default on GNU)
<code>--no-preserve-root</code>	Dangerous override — do not use casually
<code>--interactive=WHEN</code>	never, once, always

Safety

Never experiment with `rm -rf /`, `rm -rf /*`, or unquoted root globs as root.

Prefer:

- Dry-run with `find ... -print` before `-delete`

- Move to a quarantine directory first
- `trash-cli` / desktop trash when available
- Snapshots (btrfs/zfs/LVM) for real recovery options

Do not rely on alias `rm='rm -i'` alone — **scripts and other shells bypass aliases.**

Examples with Explanations

Files

```
rm file.txt
rm -v file1 file2
rm -f stale.lock           # no error if missing
```

Interactive

```
rm -i *.log
rm -I -r build/           # single confirmation for recursive
```

Directory trees

```
rm -r build/
rm -rf build/             # no prompts; double-check path first
```

Empty directory only

```
rm -d empty_dir          # fails if not empty
rmdir empty_dir          # clearer intent
```

Dry-run then delete with find

```
find ./tmpdir -type f -name '*.tmp' -print
find ./tmpdir -type f -name '*.tmp' -delete
# or interactive:
find ./tmpdir -type f -name '*.tmp' -ok rm {} \;
```

Stay on one filesystem

```
sudo rm -r --one-file-system /mnt/usb/data
```

Avoids descending into unexpected bind mounts.

Quarantine pattern

```
mkdir -p ~/QUARANTINE
mv suspect_dir ~/QUARANTINE/
# later, after review:
rm -rf ~/QUARANTINE/suspect_dir
```

Deleted but space held open

```
rm huge.log
# space may not free until writers close the fd
sudo lsof +L1 | head
sudo lsof | grep '(deleted)'
```

Notes / Pitfalls

- Globbs expand in the **shell** before **rm** runs — **rm *** is only as safe as your **cwd**.
- **rm *** skips dotfiles; **rm -r dir** removes dotfiles **inside** **dir**.
- Write permission on the **parent directory** is required to unlink a name (sticky bit on **/tmp** restricts deleting others' files).
- On CoW/SSD, **shred** is unreliable for sanitization; use disk encryption and secure erase procedures instead.
- Root can remove almost anything — accidental paths under **/var** or **/home** are common outages.

2026-relevant notes

- Prefer immutable infrastructure and snapshots over hoping **rm** is reversible.
- Container layers: deleting in a running container does not shrink the image; rebuild layers properly.
- For selective cleanup, **find/fd** + review beats recursive force from the wrong **cwd**.

Related Commands

- `unlink` — remove a single name
- `rmdir` — empty directories only
- `find -delete` — selective tree deletes
- `trash-put` — soft delete if installed
- `shred` — overwrite content (limited effectiveness)
- `lsof` — find deleted-but-open files

Additional Resources

- `man rm`

cat

Overview

`cat` (concatenate) reads files sequentially and writes them to standard output. Use it to print small files, join files, and write heredocs. For large files, prefer a pager (`less`). For syntax-highlighted viewing, consider `bat`. For following growth, use `tail -f / journalctl -f`.

Classic Unix joke still holds: `cat file | something` is often better as `something < file` or `something file`.

Syntax

```
cat [options] [file ...]
```

With no file (or `-`), reads standard input.

Common Options

Option	Description
<code>-n</code>	Number all output lines
<code>-b</code>	Number non-blank lines
<code>-s</code>	Squeeze multiple blank lines
<code>-A</code>	Show all non-printing (<code>-vET</code>)
<code>-E</code>	Show <code>\$</code> at end of lines
<code>-T</code>	Show tabs as <code>^I</code>
<code>-v</code>	Show non-printing characters
<code>-u</code>	Unbuffered (historical; often default)
<code>--</code>	End of options

Key Use Cases

1. Print small config/text files
2. Concatenate parts into one file
3. Create files via heredoc
4. Reveal invisible characters (`-A`)

5. Quick copy via redirect

Examples with Explanations

Print files

```
cat file.txt
cat /etc/os-release
cat -n file.txt          # number lines
cat -A file.txt          # show tabs/line endings
```

Concatenate

```
cat part1 part2 part3 > whole
cat part1 part2 >> whole    # append
cat header.json body.json > combined.json
```

Heredocs

```
cat <<'EOF' > greeting.txt
hello
world
EOF

cat <<EOF >> /etc/hosts
# added by bootstrap
10.0.0.5 app.local
EOF
```

Quoted 'EOF' disables expansion; unquoted EOF expands `$vars` and command substitutions.

Here-string and stdin

```
cat <<< 'single line'
cat - <<'EOF' | ssh host 'cat > /tmp/x'
content
EOF
```

Show non-printing / DOS endings

```
cat -A dosfile.txt
# CRLF shows as ^M$
```

Useless use of cat — avoid

```
# avoid
cat file | grep pattern
# prefer
grep pattern file
grep pattern < file
```

Binary caution

```
cat binary.dat          # can mess up terminal
cat binary.dat | xxd | less # better
```

Multiple files with separators (manual)

```
for f in *.conf; do
    echo "=====$f===="
    cat "$f"
done | less
```

Notes / Pitfalls

- Large files: use `less`, `tail`, or streaming tools; don't dump multi-GB logs with `cat`.
- `cat file1 file2 > file1` **truncates file1 first** — data loss. Write to a new name.
- Terminal corruption from binary output: `reset` or `tput reset`.
- Order matters for concatenation; globs are sorted by shell locale.
- BusyBox `cat` may support fewer flags.

2026-relevant notes

- Prefer `bat` for interactive reading; keep `cat` for scripts and POSIX pipelines.
- `systemd-cat` sends stdin to the journal — different tool for logging.
- For cloud-init / config blobs, heredoc with `cat <<'EOF'` remains standard in shell provisioning.

Related Commands

- `less` / `more` — pagers
- `bat` — enhanced viewing
- `tac` — reverse line order
- `head` / `tail` — portions of files
- `tee` — copy stdin to file and stdout
- `paste` / `join` — columnar combine

Additional Resources

- `man cat`

less

Overview

less is a **pager**: it displays text one screen at a time with searching, scrolling both ways, and optional follow mode. It is the default pager for **man** on most systems and a better interactive viewer than **more** for large logs and command output.

Unlike editors, **less** does not modify files (unless you explicitly open an editor from it).

Syntax

```
less [options] [file ...]  
command | less
```

Common Options

Option	Description
-N	Show line numbers
-i / -I	Ignore case in searches (smart / full)
-S	Chop long lines (no wrap); horizontal scroll
-R	Raw ANSI colors (for colored logs/ rg / ls)
-F	Quit if content fits on one screen
-X	Don't clear screen on quit (behavior depends on term)
-n	Suppress line numbers (default)
+F	Start in follow mode (like tail -f)
+G	Start at end of file
+/pattern	Start at first match
-M / -m	Long / short prompt with position
-J	Status column for marks
--incsearch	Incremental search (newer less)

Essential keys

Key	Action
q	Quit
Space / b	Page down / up
d / u	Half page down / up
j / k or arrows	Line down / up
g / G	Top / bottom
/pattern	Search forward
?pattern	Search backward
n / N	Next / previous match
&pattern	Show only matching lines
F	Follow (tail -f style); Ctrl-C to stop
ma	Mark position a
'a	Go to mark a
s	Save input to a file (when from pipe, if supported)
h	Help
v	Open in \$VISUAL / \$EDITOR

Examples with Explanations

Open files

```
less /var/log/syslog
less -N /etc/ssh/sshd_config
less +G /var/log/kern.log          # jump to end
less +/Listen /etc/ssh/sshd_config # jump to pattern
```

Pipe command output

```
journalctl -u nginx -n 500 | less
dmesg --color=always | less -R
ls --color=always | less -R
ps auxf | less
```

Follow a growing log

```
less +F /var/log/syslog
# or open normally, then press F
# Ctrl-C leaves follow mode but keeps less open
```

Chop long lines (tables, wide logs)

```
less -S /var/log/app.log
# use arrow keys / horizontal scroll
```

Case-insensitive search

```
less -i messages.log
# then /error matches Error ERROR error
```

Multiple files

```
less *.conf
# :n next file, :p previous file
```

Sensible defaults via environment

```
export LESS='-R -i -M'
export LESSHISTFILE=-          # disable ~/.lesshst if desired
export PAGER=less
export MANPAGER='less -R'
```

Compare with more / bat

```
more file.txt          # limited backward motion
less file.txt           # full interactive pager
bat file.txt            # syntax highlight; still may use less as pager
```

Notes / Pitfalls

- Without `-R`, ANSI color sequences show as junk.
- Binary files: `less` may warn; use `less -a / strings / xxd` instead of paging binaries casually.
- Very large files: `less` is efficient but searching huge unindexed logs still costs I/O.
- `LESSOPEN` preprocessor can auto-decompress `.gz` on some setups — know your distro defaults.
- When used as `PAGER`, interactive keys must work; avoid forcing `cat` as pager except in scripts.

2026-relevant notes

- `journalctl` integrates with pagers; `SYSTEMD_PAGER=less` / `PAGER=less` still apply.
- Colored tooling (`rg`, `fd`, `eza`, `ip -c`) should be paired with `less -R` or `unbuffer`-style hacks when piping.
- `bat` / `delta` improve reading UX but often shell out to `less` underneath.

Related Commands

- `more` — simpler historical pager
- `most` — alternate pager (if installed)
- `bat` — cat clone with highlighting
- `tail -f` / `journalctl -f` — follow only
- `view` / `vim -R` — read-only editor as pager
- `man` — uses a pager by default

Additional Resources

- `man less`, press `h` inside `less`

more

Overview

`more` is a simple historical **pager** that displays text one screen at a time. On modern Linux, **less** is **almost always the better interactive choice** (bi-directional scroll, better search). `more` remains useful for POSIX familiarity, tiny environments, and scripts that expect the classic pager name.

Syntax

```
more [options] [file ...]  
command | more
```

Common Options

Option	Description
<code>-d</code>	Prompt with more helpful “press space” text
<code>-f</code>	Count logical lines (don’t fold long lines as multiple)
<code>-l</code>	Do not treat form-feed specially
<code>-c / -p</code>	Paint from top / clear then show (terminal-dependent)
<code>-s</code>	Squeeze multiple blank lines
<code>-u</code>	Suppress underlining
<code>+num</code>	Start at line number
<code>+/pattern</code>	Start at pattern
<code>-n num</code>	Screen size (lines)

Feature sets differ between implementations (util-linux vs BSD heritage).

Common keys

Key	Action
Space	Next page
Enter	Next line

Key	Action
q	Quit
/pattern	Search (implementation-dependent)
h	Help (if supported)
b	Back (often not available in classic more)

Limited backward movement is the classic reason people switch to **less**.

Examples with Explanations

Basic

```
more /etc/os-release
more +20 /var/log/dmesg
more +/failed /var/log/syslog
```

Pipelines

```
dmesg | more
seq 1 200 | more
ls -l /usr/bin | more
```

Prefer less when available

```
# interactive systems
less /var/log/syslog

# force more only if required
PAGER=more man ls
```

Start position

```
more +100 /var/log/kern.log
more +/error app.log
```

Tiny environments

```
# busybox / recovery shells may only have more
more /etc/passwd
```

Compare pagers

```
printf '%s\n' {1..200} | more
printf '%s\n' {1..200} | less
```

Notice bidirectional movement and search comfort in **less**.

Notes / Pitfalls

- Default PAGER / MANPAGER on desktop distros is usually **less**, not **more**.
- Behavior of flags is **less portable** than many coreutils tools.
- Not ideal for huge logs; still reads sequentially.
- Some **more** versions exit automatically at EOF when not a tty — good for scripts, surprising interactively.
- Teaching material still mentions **more**; production muscle memory should favor **less**.

2026-relevant notes

- Recovery initramfs and busybox environments may ship **more** only — know both.
- **systemctl** / **journalctl** respect **\$SYSTEMD_PAGER** / **\$PAGER**; set to **less -R** for color.
- Don't write new tooling that requires **more**-specific keybindings.

Related Commands

- **less** — preferred full-featured pager
- **most** — alternate pager
- **bat** — highlighting viewer
- **pg** — historical pager (rare)
- **cat** — dump entire file

Additional Resources

- `man more`

chmod

Overview

chmod changes file mode bits (permissions) and special bits (setuid, setgid, sticky). Modes can be **symbolic** (`u+x`, `go-rwx`) or **octal** (755, 640). Ownership is separate — see `chown`. ACLs (`setfacl`) can grant additional access beyond the classic rwx triad.

Syntax

```
chmod [options] mode[,mode]... file...
chmod [options] --reference=RFILE file...
```

Common Options

Option	Description
-R, --recursive	Recurse into directories
-c	Report only when a change is made
-v	Verbose every file
-f	Suppress most errors
--reference=RFILE	Copy mode from reference file
-h	Affect symlinks themselves where supported

Mode primer

Octal digit	rwx meaning
7	rwx
6	rw-
5	r-x
4	r-
0	—

Common whole modes:

Mode	Typical use
755	Directories / public executables
644	Normal files
600	Private keys / secrets
700	Private directories
640	Group-readable configs
4755	setuid binary (rare; high risk)
1777	Sticky world-writable dir (/tmp style)

Symbolic form: **who** + **op** + **perms**

- who: **u** (user), **g** (group), **o** (others), **a** (all)
- op: **+** add, **-** remove, **=** set exactly
- perms: **r**, **w**, **x**, **X**, **s**, **t**

Safety

- Recursive `chmod -R 777` is almost always wrong on multi-user systems.
- Avoid `setuid`/`setgid` unless you fully understand the trust model.
- Directories need the execute bit to be entered (**x** on dirs = traverse).
- Prefer `find -type f/d` with separate modes over blind recursive octal.

Examples with Explanations

Octal

```
chmod 644 README.md
chmod 755 scripts/run.sh
chmod 600 ~/.ssh/id_ed25519
chmod 700 ~/.ssh
chmod 640 /etc/myapp/app.conf
```

Symbolic

```
chmod u+x tool.sh
chmod go-rwx secret.txt
chmod a+r file.txt
chmod u=rwx,g=rx,o= file
chmod g+s shared_dir      # setgid directory
chmod +t /shared/uploads  # sticky
```


Capital X (conditional execute)

```
chmod -R a+X project/          # add x to dirs and already-executable files
```

Useful to make trees traversable without making every file executable.

Recursive the safe way

```
find ./public -type d -exec chmod 755 {} +
find ./public -type f -exec chmod 644 {} +
```

Reference mode

```
chmod --reference=good.sh new.sh
```

Diagnose

```
ls -l file
stat -c '%a %A %n' file
namei -l /path/to/file
```

umask interaction (creation, not chmod)

```
umask
umask 022
touch newfile          # typically 644
mkdir newdir           # typically 755
```

chmod sets absolute/symbolic targets; umask affects default creation masks.

Notes / Pitfalls

- Execute on directories: without **x**, you cannot **cd** into them even if **r** is set.
- **noexec** mount option blocks execution regardless of mode.
- ACLs may allow access that **ls -l** mode bits alone don't explain — check **getfacl**.
- Copying with **cp** without **-p/-a** may drop modes; **install -m** is explicit.
- Root can chmod almost anything; wrong recursive mode on **/var/lib** causes outages.

2026-relevant notes

- Secrets: prefer 600/640 and dedicated system users; consider systemd `LoadCredential=` instead of world-readable files.
- Containers: mode bits still matter for processes sharing volumes; UIDs may be remapped.
- Git tracks executable bit only (100755 vs 100644), not full Unix modes — deploy with `install/chmod` in packaging.

Related Commands

- `chown` / `chgrp` — ownership
- `umask` — default create mask
- `stat` — numeric mode
- `getfacl` / `setfacl` — ACLs
- `install -m` — copy with mode
- `ls -l` — human view
- `namei -l` — path component modes

Additional Resources

- `man chmod`
- `info coreutils 'File permissions'`

chown

Overview

chown (change owner) sets the **user** and/or **group** ownership of files and directories. Only root (or a process with **CAP_CHOWN**) can change the user owner on typical Linux systems. Changing the group may be allowed if you are a member of the target group (rules vary; root always can).

Ownership is central to discretionary access control together with **chmod** modes and ACLs.

Syntax

```
chown [options] OWNER[:GROUP] file...
chown [options] :GROUP file...
chown [options] --reference=RFILE file...
```

Common Options

Option	Description
-R, --recursive	Operate on directory trees
-H / -L / -P	Symlink traversal policy with -R
-h	Affect symlink itself, not target (when supported)
-c	Report only when a change is made
-v	Verbose
-f	Suppress most error messages
--from=CURRENT	Change only if currently owned by CURRENT
--reference=RFILE	Copy ownership from RFILE
--preserve-root	Do not recurse on / (default on GNU)

OWNER:GROUP forms

Form	Meaning
user	Set owner; group unchanged
user:group	Set owner and group

Form	Meaning
<code>user:</code>	Set owner; group to user's primary group (GNU)
<code>:group</code>	Set group only (like <code>chgrp</code>)
numeric 0:0	UID:GID by number

Examples with Explanations

Basic

```
sudo chown alice file.txt
sudo chown alice:alice file.txt
sudo chown alice:devs project/
sudo chown :devs project/      # group only
sudo chown 1000:1000 file.txt  # numeric
```

Recursive web tree

```
sudo chown -R www-data:www-data /var/www/html
sudo chown -R alice:alice /home/alice/proj
```

Reference file

```
sudo chown --reference=/var/www/html/index.html /var/www/html/app.php
```

Conditional change

```
sudo chown --from=root:root alice:alice file.txt
```

Useful in scripts that should not clobber unexpected owners.

Symlinks

```
sudo chown -h alice linkname      # change the link's owner if supported
sudo chown -R -H alice dir_with_links
```

Know whether you mean the symlink or its target; mistakes reassign the wrong object.

Deploy pattern

```
sudo install -o root -g root -m 644 app.conf /etc/myapp/app.conf
sudo chown -R appuser:appuser /var/lib/myapp
```

Often `install/rsync --chown` is cleaner than `copy + chown`.

Audit before change

```
find /srv/data -not -user app -o -not -group app
sudo find /srv/data -not -user app -exec chown app:app {} +
```

Notes / Pitfalls

- Recursive `chown -R` on the wrong path is a classic outage (`/` or `/usr`). Prefer absolute paths and double-check.
- NFS `root_squash` maps `root` to `nobody` — remote `chown` may fail or not stick as expected.
- Containers: numeric UIDs inside user namespaces map differently on the host.
- Changing ownership does not change mode bits; pair with `chmod/setfacl` as needed.
- Sticky directories and special bits behave independently of owner changes.

2026-relevant notes

- systemd services: prefer `User= / Group=` in units and `StateDirectory=` ownership over ad-hoc `chown` in scripts.
- Rootless Podman/Docker: host UID remapping means `chown` inside may show as high UIDs on the host.
- For bulk sync with ownership, `rsync -a --chown=user:group` is often safer and more visible.

Related Commands

- `chmod` — change mode bits
- `chgrp` — change group only
- `ls -l / stat` — inspect ownership
- `id` — your uid/gids
- `install` — copy with owner/mode
- `setfacl / getfacl` — ACLs
- `namei -l` — path component owners

Additional Resources

- man chown

stat

Overview

stat displays detailed **inode metadata**: size, ownership, mode, timestamps, device/inode numbers, links, and filesystem identity. It is more precise than **ls -l** for scripting and debugging (“why can’t I write this?”, “is this the same file?”, “when was mtime?”).

GNU **stat** supports custom output formats with **-c** (files) and **-f** (filesystems).

Syntax

```
stat [options] file...
stat -c FORMAT file...
stat -f -c FORMAT file...      # filesystem (GNU)
```

Common Options

Option	Description
-c FORMAT, --format=FORMAT	Custom file format string
-f	Filesystem status instead of file
-L	Follow symlinks
-t	Terse form
--printf=FORMAT	Like -c with escape sequences; no forced newline
-x	BSD-style verbose (some systems)

Useful format sequences (GNU file)

Seq	Meaning
%n	File name
%s	Size in bytes
%a / %A	Octal / human access rights
%u / %U	UID / user name
%g / %G	GID / group name

Seq	Meaning
%i	Inode number
%d / %D	Device (decimal / hex)
%h	Hard link count
%F	File type string
%x / %X	atime human / epoch
%y / %Y	mtime human / epoch
%z / %Z	ctime human / epoch
%w / %W	birth time human / epoch (if supported)
%N	Quoted name with symlink target

Examples with Explanations

Default view

```
stat /etc/hosts
stat -L /usr/bin/python3
```

Script-friendly fields

```
stat -c '%a %n' file
stat -c '%U:%G %a %n' file
stat -c '%s bytes mtime=%y %n' file
stat -c '%d:%i %n' file          # device:inode identity
```

Compare identities (same file?)

```
stat -c '%d:%i' a b
# equal device+inode → hard-linked or same path
```

Timestamps

```
stat -c 'atime=%x' file
stat -c 'mtime=%y' file
stat -c 'ctime=%z' file
stat -c 'birth=%w' file      # may show '-' if unsupported
stat -c '%Y' file           # mtime epoch for sorting
```


Filesystem info

```
stat -f /var
stat -f -c '%T %b %f %S' /          # type, blocks, free, block size (see man)
df -T /var
```

Batch

```
stat -c '%08a %s %n' /etc/*.conf
find . -type f -printf '%' -exec stat -c '%Y %n' {} + | sort -n | tail
```

Permissions debugging

```
namei -l /var/lib/myapp/data.db
stat -c '%A %U %G %n' /var/lib/myapp /var/lib/myapp/data.db
```

Terse / portable caution

```
stat -t file                        # compact; format not for portable scripts
```

Prefer explicit `-c` formats in automation.

Notes / Pitfalls

- **ctime** is metadata change time, not “creation time”. Birth/creation is **%w** where supported (ext4, btrfs, ...).
- atime may be unreliable under **relatime/noatime** mounts.
- Following symlinks (**-L**) vs not changes which inode you inspect.
- BSD **stat** format flags differ completely — GNU examples are Linux-oriented.
- Parsing default multi-line output is fragile; always use **-c/--printf** in scripts.

2026-relevant notes

- Still the best quick metadata tool before reaching for Python **os.stat**.
- Network FS may fake or delay some timestamps; verify on the server.
- Combine with **find -printf** for bulk metadata without one **stat** exec per file when possible.

Related Commands

- `ls -l` — human listing
- `file` — content type guess
- `stat` filesystem vs `df` / `findmnt`
- `readlink` / `realpath` — path resolution
- `getfacl` — ACLs beyond mode bits
- `lsattr` / `chattr` — ext attributes

Additional Resources

- `man stat`
- [GNU coreutils — stat](#)

file

Overview

`file` classifies files by inspecting **content magic** (and optionally the filesystem type), not just the name extension. Use it on unknown downloads, mystery binaries, compressed blobs, and scripts without a reliable extension.

It reads magic patterns from a database (e.g. `/usr/share/misc/magic`). Classification is heuristic — good for ops triage, not a security sandbox.

Syntax

```
file [options] file...
file -                # read stdin
```

Common Options

Option	Description
<code>-b, --brief</code>	Don't prefix filename
<code>-i, --mime</code>	MIME type output
<code>-I</code>	MIME with encoding (some versions)
<code>-z</code>	Look inside compressed files
<code>-L</code>	Follow symlinks
<code>-s</code>	Allow special/block files
<code>-k</code>	Keep going; don't stop at first match
<code>-f listfile</code>	Read filenames from listfile
<code>-e test</code>	Exclude a test (e.g. <code>soft</code> , <code>tokens</code>)
<code>-h</code>	Don't follow symlinks (default often)

Examples with Explanations

Basic classification

```
file /bin/ls
file /etc/hosts
file mystery.bin
file photo.jpg note.txt archive.tar.gz
```

Brief and MIME

```
file -b mystery.bin
file -i photo.jpg
# photo.jpg: image/jpeg; charset=binary
file -bi photo.jpg
```

Inside compressed data

```
file -z backup.tar.gz
file -z something.xz
```

Symlinks

```
file /usr/bin/python3
file -L /usr/bin/python3
```

Scripts and interpreters

```
file bootstrap.sh
# may report: Bourne-Again shell script, ASCII text executable
head -1 bootstrap.sh      # shebang check
```

Stdin

```
head -c 256 blob | file -
curl -fsSL https://example.com/file | file -
```

Batch unknown directory

```
file *
find . -type f -print0 | xargs -0 file | grep -i 'executable\|ELF'
```

Guard uploads / pipelines

```
mime=$(file -bi "$upload" | cut -d';' -f1)
case $mime in
  image/jpeg|image/png) echo ok ;;
  *) echo "rejected: $mime" >&2; exit 1 ;;
esac
```

ELF deep dive (after file)

```
file /usr/local/bin/app
readelf -h /usr/local/bin/app
ldd /usr/local/bin/app
```

Notes / Pitfalls

- Extensions lie; `file` can too — polyglots and crafted headers fool magic.
- Text encodings may be guessed; don't treat as authoritative Unicode detection.
- Very large files: `file` only needs the beginning — still be careful with special devices.
- Without `-s`, behavior on device nodes is limited.
- Magic DB age depends on package `libmagic` / `file` updates.

2026-relevant notes

- Container distroless images may omit `file`; copy samples to a debug image.
- For security scanning, use dedicated malware/YARA tooling beyond `file`.
- MIME output pairs well with web and S3 content-type checks in shell tooling.

Related Commands

- `stat` — metadata, not content type
- `hexdump` / `xxd` — raw bytes
- `strings` — printable spans in binaries
- `readelf` / `objdump` — ELF analysis

- `identify` (ImageMagick) — image details
- `ls -l` — names and modes only

Additional Resources

- `man file`
- [libmagic / file\(1\) docs](#)

find

Overview

`find` walks a directory tree and selects files by name, type, time, size, permissions, and more. It can also **act** on matches (`-exec`, `-delete`). Prefer it for precise searches; use `locate` only for pre-indexed name lookups.

Syntax

```
find [path...] [expression]
```

Expression primaries are AND by default; use `-o`, `\!`, and escaped parentheses `\( \)` for complex logic.

Common Options / Primaries

Primary / option	Description
<code>-name PAT / -iname PAT</code>	Basename glob (case-insensitive with <code>-iname</code>)
<code>-path PAT / -ipath PAT</code>	Full path glob
<code>-regex PAT</code>	Path regex (see <code>-regextype</code>)
<code>-type f\ d\ l\ b\ c\ p\ s</code>	File type
<code>-size n[cwbkMG]</code>	Size (+ <i>n</i> greater, - <i>n</i> less)
<code>-mtime n / -mmin n</code>	Modified <i>n</i> days / minutes ago
<code>-atime / -ctime</code>	Access / inode change time
<code>-user / -group</code>	Owner
<code>-perm mode</code>	Permissions (<code>-perm -u+x</code> , <code>-perm /u=x</code>)
<code>-empty</code>	Empty file or directory
<code>-maxdepth n / -mindepth n</code>	Limit descent
<code>-xdev</code>	Do not cross filesystems
<code>-prune</code>	Do not descend into match
<code>-print0</code>	NUL-delimited paths (safe for weird names)
<code>-exec cmd {} \;</code>	Run per file
<code>-exec cmd {} +</code>	Batched args (like <code>xargs</code>)
<code>-delete</code>	Delete match (implies depth-first)

Key Use Cases

1. Locate files by name, age, or size across trees
2. Clean temp/log artifacts safely
3. Feed paths into greps, compressors, or fixers
4. Audit permissions and setuid files
5. Stay on one mount with `-xdev` during disk hunts

Safety

`-delete` and `-exec rm` are irreversible. Test with `-print` first. Quote globs so the shell does not expand them (`-name '*.log'`). Prefer `-exec ... {} +` or `-print0 | xargs -0` for speed and safety with spaces.

Examples with Explanations

Find by name (case-insensitive)

```
find /var/log -iname '*.log' -type f
```

Basename match only; still recurses into subdirectories.

Limit depth

```
find /etc -maxdepth 2 -type f -name '*.conf'
```

Avoids walking huge trees when you know the shape.

Large files on one filesystem

```
find / -xdev -type f -size +500M -printf '%s\t%p\n' 2>/dev/null | sort -nr | head
```

`-xdev` skips other mounts (NFS, USB). `2>/dev/null` hides permission noise.

Modified in the last day

```
find ~/projects -type f -mtime -1
```

`-mtime -1` = less than 24 hours ago (integer day semantics; see also `-mmin`).

Exclude a directory

```
find . -path './.git' -prune -o -path './node_modules' -prune -o -type f -name '*.js' -print
```

`-prune` skips those trees; structure is: `prune OR (criteria AND print)`.

Safe batch action

```
find /var/tmp -type f -mtime +30 -print0 | xargs -0 -r rm -v
```

NUL-delimited pipeline handles spaces; `-r` skips `rm` if empty.

Exec with batching

```
find . -type f -name '*.png' -exec gzip -9 {} +
```

One `gzip` with many args — faster than `\;`.

Permission audit (setuid)

```
find /usr -xdev -perm -4000 -type f -printf '%m %u %p\n' 2>/dev/null
```

Classic security inventory.

Empty directories

```
find . -type d -empty
```

Understanding Output

Default is one path per line (`-print`). Use `-printf` for custom columns (size, mtime, mode). Exit status is non-zero if errors occurred during traversal (e.g. permission denied), even if some matches printed.

Common Usage Patterns

Newest N files under a tree

```
find . -type f -printf '%T@ %p\n' | sort -nr | head | cut -d' ' -f2-
```

Delete with interactive dry-run

```
find /tmp -type f -mtime +14 -print    # review
find /tmp -type f -mtime +14 -delete  # then delete
```

Notes & Pitfalls

- Day filters (`-mtime`) truncate; for “last 2 hours” use `-mmin -120`.
- Globs in `-name` are **not** regex; use `-regex` for regular expressions.
- `find -delete` deletes directories only when empty (after contents removed in depth-first order).
- Prefer `locate` for “I only know a filename and the DB is fresh”; prefer `find` for live criteria.

Related Commands

- `locate` / `updatedb` — indexed name search
- `fd` — modern friendlier finder (if installed)
- `grep -R` — search *contents*
- `du` / `ncdu` — size-oriented exploration
- `xargs -0` — build argument lists from `-print0`

Additional Resources

- `man find`
- [GNU Findutils manual](#)

fd

Overview

fd is a fast, user-friendly alternative to **find**. Defaults are developer-friendly: colorized output, sensible ignore of hidden files and **.gitignore** entries, parallel traversal, and simple regex patterns. Keep **find** for full POSIX power and exotic predicates; use **fd** for everyday “find me files named ...” work.

```
sudo apt install fd-find    # Debian/Ubuntu binary often named fdfind
# symlink tip on Debian:
# mkdir -p ~/.local/bin && ln -s $(command -v fdfind) ~/.local/bin/fd
sudo dnf install fd-find
```

Syntax

```
fd [options] [pattern] [path...]
```

Common Options

Option	Description
-e EXT	Extension filter (repeatable)
-t f/d/l/x/e	Type: file, dir, symlink, executable, empty
-H	Include hidden
-I	Don't respect ignore files
-u / -uu	Unrestricted (like rg's -u)
-x cmd	Exec per result ({} placeholder)
-X cmd	Batch exec with all results
-d N	Max depth
-g glob	Glob mode instead of regex
-s	Case sensitive (default may be smart case)
-p	Match full path
-a	Absolute paths
-0	NUL-delimited
-S size	Size filter (+1m, -100k)
--changed-within	Time filters (version-dependent)

Key Use Cases

1. Find files by name/extension quickly
2. Run a command on each match (-x)
3. Search including gitignored build dirs (-I / -u)
4. Generate file lists for pipelines (-0 + xargs -0)

Examples with Explanations

By extension and pattern

```
fd -e md
fd -e go -e rs
fd 'config.*\ya?ml$'
fd -g '*.toml'
```

Types and hidden

```
fd -t d node_modules
fd -t f -e log /var/log
fd -H '^\.env'
```

Exec

```
fd -e log -x rm
fd -e jpg -x convert {} {}.thumb.jpg # if ImageMagick present
fd -e md -X wc -l
```

-x runs once per file; -X batches arguments.

Ignore rules

```
fd pattern          # respects .gitignore
fd -I pattern        # include ignored
fd -u secret         # more unrestricted
```

One-liner recipes

```
# Recent-ish markdown under repo (if --changed-within available)
fd -e md --changed-within 7d

# Safe delete of __pycache__
fd -t d -g __pycache__ -x rm -rf

# Feed ripgrep
fd -e py -0 | xargs -0 rg 'TODO'
```

Notes & Pitfalls

- On Debian/Ubuntu the binary is often **fdfind** to avoid clashing with **fdclone**.
- Regex is **not** shell glob by default—use **-g** for globs.
- **-x rm** without care deletes widely; preview with plain **fd** first.
- Color can break scripts—use **fd --color never** or redirect.

2026-relevant notes

- **fd + rg** is the modern find/grep pair for source trees; **find + grep** remain the portable baseline.
- Large monorepos: prefer **fd**'s parallelism; for network filesystems test both tools.
- Combine with **fzf** for interactive picking: **fd -t f | fzf**.

Comparison to alternatives

Tool	Notes
fd	Fast UX defaults
find	Full predicates, portable
locate/plocate	DB-backed name search
rg --files	List files rg would search

Related Commands

- **find** — classic recursive search
- **rg** — content search
- **locate** — indexed names
- **tree / eza -T** — visual trees

Additional Resources

- [Upstream fd README](#)
- `man fd` / `man fdfind`

locate

Overview

`locate` finds files by **name** using a prebuilt database (historically `mlocate`; on modern Ubuntu often `plocate`). It is much faster than walking the live filesystem with `find` for simple name queries, but results can be **stale** until the database is updated.

Use `find` when you need live data or predicates (size, mtime, permissions, content).

Syntax

```
locate [options] pattern...
```

Common Options

Option	Description
<code>-i</code>	Case-insensitive
<code>-l N</code> / <code>-n N</code>	Limit to N results
<code>-b</code> / <code>--basename</code>	Match basename only
<code>-r</code> / <code>--regex</code>	Extended regex
<code>-c</code>	Count matches
<code>-e</code>	Only print paths that still exist
<code>-0</code>	NUL-delimited output
<code>-d DB</code> / <code>--database=DB</code>	Alternate database path
<code>-A</code>	Match all patterns (AND; when supported)
<code>-w</code>	Match whole path as a word (implementation-dependent)

Check `man locate` / `man plocate` for your provider.

Database updates

```
sudo updatedb
# plocate often uses a systemd unit/timer:
sudo systemctl start plocate-updatedb.service
systemctl status plocate-updatedb.timer
```

Daily timers/cron usually refresh automatically. Freshly created files won't appear until the next update.

Examples with Explanations

Basic

```
locate sshd_config
locate -i readme
locate -b '\.bashrc'
locate nginx.conf
```

Limit and count

```
locate -l 20 nginx
locate -n 50 '*.service'
locate -c '*.png'
```

Regex

```
locate -r '/etc/.*\.conf$'
locate -r 'systemd.*\.service$'
```

Basename-only

```
locate -b 'passwd'
# vs full path match noise:
locate passwd | head
```


Existence filter

```
locate -e mydeletedfile      # suppress already-removed paths still in DB
```

NUL-safe handoff

```
locate -0 -b 'TODO.md' | xargs -0 ls -l
```

When locate is empty

```
command -v locate
dpkg -l plocate mlocate 2>/dev/null
sudo apt install plocate      # Debian/Ubuntu example
sudo systemctl start plocate-updatedb.service
```

locate vs find

Need	Tool
Fast name search system-wide	<code>locate</code>
Just-created files	<code>find</code> / <code>fd</code>
Size / mtime / mode filters	<code>find</code>
Content search	<code>rg</code> / <code>grep -R</code>
PATH executables only	<code>command -v</code> / <code>type</code>

Notes / Pitfalls

- Security/privacy: databases may omit unreadable paths or require permissions to query — don't assume world visibility of private files.
- Patterns are often substring matches against full paths — expect noise without `-b` or regex anchors.
- Network filesystems may be excluded from `updatedb` for performance (see `PRUNEPATHS` / `plocate` config).
- Different implementations (`mlocate` vs `plocate`) change defaults and speed; prefer distro docs.
- `locate` is not a real-time security audit of “what exists now”.

2026-relevant notes

- **plocate** is the common modern default — faster compressed DB.
- Immutable/ostree systems may index differently; containers often lack a DB entirely.
- For developer trees, **fd**/**find** inside the project beats system **locate**.

Related Commands

- **find** — live criteria search
- **fd** — modern friendly finder
- **updatedb** / **plocate-updatedb** — refresh DB
- **which** / **type** — executables on PATH
- **dpkg -S** / **rpm -qf** — which package owns a path

Additional Resources

- `man locate`, `man plocate`, `man updatedb`

ripgrep (rg)

Overview

ripgrep (rg) recursively searches the filesystem for a regex pattern. It respects `.gitignore` by default, skips hidden/binary files sensibly, and is typically much faster than `grep -r` on large codebases.

```
sudo apt install ripgrep
sudo dnf install ripgrep
```

Syntax

```
rg [options] PATTERN [PATH...]
```

Common Options

Option	Description
-i	Case insensitive
-w	Whole word
-t TYPE	File type (py, rust, go, md, ...)
-T TYPE	Exclude type
-g GLOB	Include/exclude globs (-g '!**/vendor/**')
-C N / -A / -B	Context lines
-n / -N	Line numbers on/off
-l	Files with matches
-c	Count per file
-o	Only matching part
-v	Invert match
-u / -uu / -uuu	Unrestricted (ignore ignore rules / hidden / binary)
--hidden	Search hidden files
--no-ignore	Don't respect ignore files
-F	Fixed string (no regex)
-P	PCRE2 (if built with it)
-z	Search compressed files
--json	JSON lines output

Option	Description
-0 with -l	NUL file names
--stats	Search statistics

Key Use Cases

1. Codebase search ignoring build artifacts
2. Language-scoped searches (-t go)
3. Incident hunting in logs (with -uu when needed)
4. Pre-commit / review “where is this symbol?”

Examples with Explanations

Basics

```
rg 'process_data'  
rg -i 'todo|fixme'  
rg -w 'main'
```

Types and globs

```
rg -t python 'import os'  
rg -t go -t rust 'TODO'  
rg -g '*.qmd' 'include:.'  
rg -g '!**/node_modules/**' 'left-pad'
```

Context and files

```
rg -C 3 'panic\('  
rg -l 'DEPRECATED'  
rg -c 'error' /var/log 2>/dev/null
```

Unrestricted search

```
rg -uu 'SECRET_TOKEN'  
rg --hidden --no-ignore 'password'
```

One-liner recipes

```
# Who calls this function?
rg -n 'funcName\(\' -t go

# JSON for tools
rg --json 'pattern' | head

# Replace workflow: list then edit
rg -l 'OldName' | xargs -r sed -i 's/OldName/NewName/g'    # review first!

# Fixed string with special chars
rg -F 'C++' -t md
```

Notes & Pitfalls

- Default ignore can **hide** hits in **build/**, **dist/**, or gitignored configs—use **-u** deliberately.
- Binary matches are skipped unless unrestricted; good for speed, surprising for firmware blobs.
- Multiline requires **-U** / **--multiline** (see man).
- **grep** is still required for POSIX scripts and tiny systems without **rg**.

2026-relevant notes

- **rg** + **fd** is the standard modern search pair; IDEs shell out to similar engines.
- For huge monorepos, prefer type/glob filters over unconstrained **-uuu**.
- PCRE2 (**-P**) is optional at build time—don't assume it on every distro build.

Comparison to alternatives

Tool	Role
rg	Fast recursive search
grep -R	Portable
ag / ack	Older dev search tools
git grep	Only tracked files

Related Commands

- **grep** — standard search
- **fd** — find files by name
- **git grep** — repo-tracked search

- `find ... -exec grep` — classic combo

Additional Resources

- `man rg`
- [BurntSushi/ripgrep user guide](#)

ln

Overview

ln creates **links**: hard links (default) or symbolic links (**-s**). Symlinks are path pointers and can cross filesystems and point at directories. Hard links are additional directory entries for the **same inode**, must stay on one filesystem, and generally cannot link directories on Linux.

Syntax

```
ln [options] target link_name
ln [options] target... directory
ln [options] -t directory target...
```

Common Options

Option	Description
-s, --symbolic	Create a symbolic link
-f, --force	Remove existing destination files
-i, --interactive	Prompt before removing destination
-n, --no-dereference	Treat link_name as a normal file if it is a symlink to a dir
-v, --verbose	Verbose
-r, --relative	Create symlink with relative target path (GNU)
-T	Always treat link_name as a normal file
-t DIR	Directory to place links into
-b	Backup destination before overwrite

Hard vs symbolic

Property	Hard link	Symlink
Cross filesystem	No	Yes
Point to directory	No (normal Linux)	Yes
Broken if target removed	Still has data if any link remains	Becomes dangling

Property	Hard link	Symlink
Relative semantics	N/A (inode)	Stored path can be relative or absolute
<code>ls -l</code>	Same inode, no <code>-></code>	Shows <code>-> target</code>

Examples with Explanations

Symbolic links

```
ln -s /usr/bin/python3 /tmp/py
ln -s ../config/app.yml .
ln -sfn /opt/app/releases/1.2 /opt/app/current # atomic-ish switch
```

`-f` replaces; with symlinks to dirs, combine carefully with `-n/-T` to avoid nesting into the old dir.

Relative symlink (GNU)

```
ln -sr /opt/app/shared/.env /opt/app/current/.env
```

Stores a relative path from link location to target — friendlier when trees move together.

Hard links

```
ln file.txt file-hard.txt
ls -li file.txt file-hard.txt # same inode
```

Bulk into a directory

```
ln -s /usr/bin/python3 /usr/bin/pip3 ~/bin/
ln -t ~/bin -s /usr/local/bin/app
```

Inspect

```
ls -l link
readlink link
readlink -f link
stat link
namei -l /path/to/link
```


Replace symlink safely

```
ln -sfn /opt/app/releases/1.3 /opt/app/current
# or
ln -s /opt/app/releases/1.3 /opt/app/current.new
mv -Tf /opt/app/current.new /opt/app/current
```

Common failure: relative target confusion

```
cd /tmp
ln -s file.txt link          # target is string "file.txt" relative to link's directory
```

If you create the link elsewhere, a relative target resolves from the **link's** directory, not your cwd at creation time (you stored a string).

Notes / Pitfalls

- `ln -s existing_dir name` when `name` exists as directory may create `name/existing_dir` — use `-T/-n` and check.
- Hard links to shared libraries/configs can confuse package upgrades; prefer symlinks for admin sugar.
- Backup tools and containers may treat hard links specially (`rsync -H`).
- Dangling symlinks are valid entries; tools may or may not follow them.
- Permissions on a symlink are largely ignored on Linux; the **target's** mode matters for access through the link.

2026-relevant notes

- Blue/green deploys still use **current** symlink switches extensively.
- Special filesystems (overlay, FUSE) may restrict hard links.
- Prefer `ln -sfn` patterns carefully; test once in a throwaway path.

Related Commands

- `readlink / realpath` — resolve links
- `ls -l / stat` — inspect
- `cp -a / cp -P` — copy preserving links
- `namei` — walk path components
- `find -type l` — find symlinks
- `rm` — remove a link (does not delete hard-linked data until last link)

Additional Resources

- `man ln`

readlink

Overview

`readlink` prints the target of a symbolic link, or with GNU flags like `-f/-e/-m`, canonicalizes paths by resolving symlinks (similar territory to `realpath`). Essential for debugging link chains, deploy symlinks, and locating the real path of `$0` in scripts.

Syntax

```
readlink [options] file...
```

Common Options (GNU)

Option	Description
<code>-f, --canonicalize</code>	Canonical path; all but last component must exist
<code>-e, --canonicalize-existing</code>	All components must exist
<code>-m, --canonicalize-missing</code>	No component need exist; still normalize
<code>-n, --no-newline</code>	Do not output trailing newline
<code>-q, -s</code>	Suppress most errors
<code>-v</code>	Verbose errors
<code>-z</code>	NUL-terminate output
<code>--</code>	End of options

Without flags, `readlink` prints the **raw** symlink contents (may be relative) and fails if the path is not a symlink.

Key Use Cases

1. Show immediate symlink target
2. Resolve full canonical path (`-f/-e`)
3. Script location detection
4. Verify deploy **current** links
5. Safe path comparisons

Examples with Explanations

Immediate target

```
ln -s ../etc/hosts /tmp/h
readlink /tmp/h
# ../etc/hosts
```

Canonical forms

```
readlink -f /tmp/h
readlink -e /usr/bin/python3
readlink -m /not/yet/created
```

Flag	Missing path behavior
-e	Fail if any component missing
-f	Last component may be missing (GNU)
-m	Tolerate missing components

Multiple links

```
readlink -f /usr/bin/python /usr/bin/java
```

Script directory pattern

```
#!/usr/bin/env bash
SELF=$(readlink -f "$0")
ROOT=$(dirname "$SELF")
cd "$ROOT"
```

BusyBox / portability note

```
# portable-ish alternative when readlink -f missing:
realpath "$(dirname "$0")"
# or
python3 -c 'import os,sys; print(os.path.realpath(sys.argv[1]))' "$0"
```

Inspect vs ls

```
ls -l /usr/bin/python3
readlink /usr/bin/python3
readlink -f /usr/bin/python3
```

NUL-safe with find

```
find /opt -type l -print0 | while IFS= read -r -d '' L; do
    printf '%s -> %s\n' "$L" "$(readlink "$L")"
done
```

Notes / Pitfalls

- Bare **readlink** **errors** on non-symlinks; **readlink -f** works on regular files too (returns canonical path).
- Relative symlink text is relative to the **link's directory**, not your cwd.
- **readlink -f** behavior differs slightly across BSD/GNU; check man pages on non-Linux.
- Prefer **realpath** when you always want canonicalize semantics by name clarity.
- Race: target can change between resolve and use — TOCTOU in security-sensitive code.

2026-relevant notes

- Deploy tooling still relies on **readlink -f** for resolving release symlinks.
- In containers, resolved paths reflect container mounts, not host paths.
- Combine with **namei -l** when permissions on intermediate components fail.

Related Commands

- **realpath** — canonicalize paths
- **ln -s** — create symlinks
- **ls -l** — show link arrow
- **stat** — inode metadata
- **namei** — stepwise path resolution
- **dirname / basename** — split paths

Additional Resources

- `man readlink`

realpath

Overview

`realpath` prints the resolved absolute path of each argument, expanding `.` / `..` and optionally resolving symbolic links. Use it for canonical comparisons, stable script roots, and turning relative paths into absolute ones without reinventing path logic in bash.

GNU `realpath` overlaps with `readlink -f/-e/-m` but is clearer when “canonicalize this path” is the intent.

Syntax

```
realpath [options] file...
```

Common Options

Option	Description
<code>-e, --canonicalize-existing</code>	All components must exist
<code>-m, --canonicalize-missing</code>	No component need exist
<code>-s, --no-symlinks</code>	Do not expand symlinks
<code>-q, --quiet</code>	Suppress most error messages
<code>-z, --zero</code>	NUL-terminated output
<code>--relative-to=DIR</code>	Print path relative to DIR
<code>--relative-base=DIR</code>	Absolute unless under base
<code>-L / -P</code>	Logical / physical variants (see man)

Default (no `-e/-m`) is similar to canonicalizing with last component requirements depending on version — prefer explicit `-e` or `-m` in scripts.

Key Use Cases

1. Absolute canonical paths
2. Relative path between two locations
3. Script root detection

4. Compare whether two path strings name the same place
5. Normalize user input paths

Examples with Explanations

Absolute paths

```
realpath ../file.txt
realpath -e /etc/hosts
realpath -m /not/yet/created
realpath -m ./build/out/binary
```

Relative formatting

```
realpath --relative-to=/home/user /home/user/docs/file.txt
# docs/file.txt
realpath --relative-to="$PWD" /etc/hosts
realpath --relative-base=/srv /srv/app/bin/tool
```

Script root

```
#!/usr/bin/env bash
set -euo pipefail
ROOT=$(realpath "$(dirname "$0")")
# or resolve the script itself first if it may be a symlink:
ROOT=$(dirname "$(realpath -e "$0")")
cd "$ROOT"
```

Same-file check

```
p1=$(realpath -e "$1")
p2=$(realpath -e "$2")
[[ $p1 == "$p2" ]] && echo 'same path'
# stronger: compare device+inode
stat -c '%d:%i' "$p1" "$p2"
```

No symlink expand

```
realpath -s /tmp/linkdir
```

Batch NUL-safe

```
printf '%s\0' ./ * | xargs -0 realpath -e
```

Combine with find

```
find . -name '*.so' -print0 | xargs -0 realpath
```

Notes / Pitfalls

- Without `-m`, missing paths error — good for validation, bad for pre-creating destinations.
- Canonicalization depends on current mounts and namespaces (containers differ from hosts).
- Race conditions apply: resolved path may change before use.
- BusyBox may offer a smaller `realpath`; test flags.
- `realpath .` vs `pwd -P` — similar idea for `cwd`.

2026-relevant notes

- Prefer `realpath` in new scripts over ad-hoc bash string chopping.
- Build systems and monorepos use relative-from-root paths extensively — `--relative-to` helps generate them.
- Network FS: resolving through automount points can trigger mounts as a side effect.

Related Commands

- `readlink -f` — GNU canonicalize via `readlink`
- `pwd -P` — physical `cwd`
- `dirname / basename` — split components
- `cd -P` — physical change directory
- `stat` — identity via inode
- `findmnt` — mount context

Additional Resources

- `man realpath`

basename

Overview

`basename` strips the directory portion (and optionally a suffix) from a path, leaving the final component. It is the complement of `dirname` for simple path surgery in shell scripts — naming logs, deriving output files, and parsing `$0`.

Syntax

```
basename NAME [SUFFIX]
basename OPTION... NAME...
```

Common Options

Option	Description
<code>-a, --multiple</code>	Treat all arguments as NAMEs
<code>-s SUF, --suffix=SUF</code>	Strip suffix (implies multiple mode when combined properly)
<code>-z, --zero</code>	NUL-terminated output

Traditional form: `basename /path/to/file.txt .txt → file`.

Key Use Cases

1. Extract filename from a path
2. Strip extensions
3. Derive names from `$0` or CLI args
4. Build sibling output paths in loops
5. Clean display names in logs

Examples with Explanations

Basics

```
basename /path/to/file.txt
# file.txt
basename /path/to/file.txt .txt
# file
basename /path/to/dir/
# dir
basename /
# / (or implementation-defined root behavior)
```

Multiple names

```
basename -a /a/b /c/d
basename -a -s .conf /etc/ssh/sshd_config /etc/systemd/logind.conf
```

Script name

```
SCRIPT_NAME=$(basename "$0")
echo "usage: $SCRIPT_NAME [options]"
```

Strip extension patterns

```
f=/data/report.2024.csv
base=$(basename "$f")           # report.2024.csv
stem=$(basename "$f" .csv)      # report.2024
# careful: only strips exact suffix match at end
```

Loop rename

```
for f in /var/log/app/*.log; do
    b=$(basename "$f")
    gzip -c "$f" > "/backup/${b}.gz"
done
```

Combine with dirname

```
path=/opt/app/bin/tool
echo "dir=$(dirname "$path") name=$(basename "$path")"
```

NUL-safe pipeline

```
find . -type f -name '*.md' -print0 |
while IFS= read -r -d '' f; do
    basename -z "$f"
done
```

Parameter expansion alternative

```
# bash-only equivalents often used instead of basename
f=/path/to/file.txt
echo "${f##*/}"          # file.txt
echo "${f##*/}" | sed 's/\.txt$//'
# or
b=${f##*/}; echo "${b%.txt}"
```

Builtins avoid spawning `basename` in hot loops.

Notes / Pitfalls

- Suffix strip requires an **exact** trailing match; `.tar.gz` must be stripped carefully (often twice or with special logic).
- Trailing slashes: GNU `basename` handles them; still quote variables.
- Don't use for security canonicalization — use `realpath` / proper validation.
- BusyBox `basename` may lack `-a/-z`.
- Filenames with newlines need `-z` / careful reading.

2026-relevant notes

- In hot path bash, prefer `${var##*/}` over external `basename`.
- Still excellent for readability in admin scripts and one-liners.
- Pair with `realpath` when the path may contain `..` or symlinks before basenaming for display only.

Related Commands

- `dirname` — parent path
- `realpath` — canonicalize
- `readlink` — resolve links
- bash parameter expansion — `${f##*/}`, `${f%.*}`
- `cut` / `sed` — ad-hoc parsing

Additional Resources

- `man basename`

dirname

Overview

`dirname` removes the last path component, returning the directory portion of a path. Together with `basename` it covers most shell path-splitting needs: locating a script's directory, ensuring parent dirs exist, and computing sibling paths.

Syntax

```
dirname [OPTION] NAME...
```

Common Options

Option	Description
<code>-z, --zero</code>	NUL-terminated output

GNU `coreutils` accepts multiple `NAME` arguments and prints one line each.

Key Use Cases

1. Find a script's directory
2. `mkdir -p` parents before writing a file
3. Compute sibling paths
4. Normalize path structure in scripts
5. Logging / display of parent locations

Examples with Explanations

Basics

```
dirname /path/to/file.txt
# /path/to
dirname file.txt
# .
dirname /usr/local/bin/
# /usr/local
dirname /usr
# /
dirname /
# /
```

Multiple arguments

```
dirname /a/b /c/d/e
```

Script directory pattern

```
#!/usr/bin/env bash
set -euo pipefail
SCRIPT_DIR=$(cd "$(dirname "$0")" && pwd)
# better when $0 is a symlink:
SCRIPT_DIR=$(cd "$(dirname "$(readlink -f "$0")")" && pwd)
cd "$SCRIPT_DIR"
```

Ensure parent exists

```
out=/var/lib/myapp/data/file.db
mkdir -p "$(dirname "$out")"
touch "$out"
```

Sibling paths

```
conf=/etc/myapp/app.conf
dir=$(dirname "$conf")
cp "$conf" "$dir/app.conf.bak"
```

Combine with realpath

```
abs=$(realpath -m "$1")
parent=$(dirname "$abs")
```

Parameter expansion alternative (bash)

```
f=/path/to/file.txt
echo "${f%/*}"          # /path/to (careful with no-slash cases)
# dirname is safer for edge cases like no slash → "."
```

NUL-safe

```
find /etc -name '*.conf' -print0 |
  xargs -0 -n1 dirname |
  sort -u
```

Notes / Pitfalls

- `dirname file` (no slash) → `.` not empty string — important for `cd`.
- Trailing slashes are normalized by GNU `dirname`.
- Don't implement security checks with string `dirname` alone; canonicalize first.
- Hot loops: `bash ${f%/*}` is faster but handle “no slash” yourself.
- `dirname` does not require the path to exist.

2026-relevant notes

- Still the readable choice in installer and devops shell scripts.
- For complex path logic, consider Python/`pathlib` in larger tools.
- Pair with `realpath -m` when creating outputs under computed absolute parents.

Related Commands

- `basename` — final component
- `realpath` — canonicalize
- `readlink -f` — resolve script path
- `mkdir -p` — create parents
- `cd / pwd` — navigate and display
- bash expansions — `${f%/*}`

Additional Resources

- `man dirname`

which

Overview

`which` locates executables by searching the `PATH` environment variable and printing the path of the first match. It is a traditional convenience for interactive use. For scripts, prefer `command -v` (POSIX) or `type -P` (bash) — they integrate better with shell functions, aliases, and builtins.

Syntax

```
which [options] command...
```

Implementations differ (GNU `which`, Debian shell script `which`, BusyBox). Options are **not** fully portable.

Common Options (GNU / common)

Option	Description
<code>-a</code>	Show all matches on <code>PATH</code> , not just the first
<code>-s</code>	Silent; exit status only (some versions)
<code>--version</code> / <code>--help</code>	Version / help (GNU)

Key Use Cases

1. Interactive “where is this binary?”
2. See which of several installs comes first (`-a`)
3. Quick sanity check after modifying `PATH`
4. Compare with shell resolution (`type`)

Examples with Explanations

Basic

```
which python3
which ls
which docker kubectl
```

All matches

```
which -a python3
which -a java
```

Shows shadowing (e.g. /usr/local/bin before /usr/bin).

Prefer command -v in scripts

```
# good for scripts
if command -v jq >/dev/null 2>&1; then
    jq --version
fi

# bash: PATH only, ignore functions/aliases
if type -P jq >/dev/null; then
    echo "jq binary found"
fi
```

Compare resolution tools

```
type python3
type -a python3
command -v python3
which python3
which -a python3
```

Tool	Sees aliases	Sees functions	Sees builtins	PATH binaries
type	yes	yes	yes	yes
command -v	yes*	yes*	yes	yes
which	often no	no	no	yes

*shell-dependent; still better than `which` for “what will run”.

Debug PATH order

```
echo "$PATH" | tr ':' '\n' | nl  
which -a node
```

Silent existence (when -s exists)

```
which -s docker && echo present  
# portable:  
command -v docker >/dev/null && echo present
```

Notes / Pitfalls

- Debian/Ubuntu historically shipped **which** as a shell script with different flags; don't rely on GNU-only options in portable scripts.
- **which** may not know about shell aliases/functions you use interactively — misleading for “what runs when I type this”.
- Hashed commands in bash (**hash -r** to clear) affect speed of lookup, not usually **which**'s external search.
- Relative PATH entries (**.** or empty) are a security footgun — **which** will show what they resolve to.
- In login vs non-login shells, PATH differs; results differ.

2026-relevant notes

- Version managers (asdf, nvm, pyenv, mise) put shims early on PATH — **which -a** is useful to see real binaries behind shims.
- Containers often have a minimal PATH; **command -v** in entrypoints is the standard check.
- Prefer documenting **command -v** in runbooks and automation.

Related Commands

- **command -v** / **command -V** — POSIX resolution
- **type** / **type -a** — bash detailed resolution
- **whereis** — binary/source/man search (rough)
- **ls -l** / **readlink -f** — inspect the binary found
- **enable** — manage bash builtins

Additional Resources

- `man which` (if present), `help type`, `help command`

alias / unalias

Overview

`alias` is a **shell builtin** that creates command shortcuts in the current shell session. Aliases are primarily for interactive convenience. They are **not** automatically available in non-interactive scripts (unless you explicitly enable/source them). For anything with arguments or logic, prefer a **shell function**.

Persist aliases in `~/.bashrc`, `~/.zshrc`, or a dedicated `~/.bash_aliases` sourced from your rc file.

Syntax

```
alias
alias name='command string'
alias name='$'command with specials'
unalias name
unalias -a
```

Common Options / related

Form	Description
<code>alias</code>	List defined aliases
<code>alias name=value</code>	Define alias
<code>unalias name</code>	Remove one
<code>unalias -a</code>	Remove all
<code>type name</code>	Show how name resolves
<code>command name</code>	Bypass alias/function for external
<code>\name</code>	Bypass alias

Examples with Explanations

Everyday shortcuts

```
alias ll='ls -lah'
alias la='ls -A'
alias gs='git status'
alias ..='cd ..'
alias ...='cd ../..'
alias grep='grep --color=auto'
alias ip='ip -c'
alias please='sudo'
```

List and inspect

```
alias
alias ll
type ll
type -a ls
```

Persist (bash)

```
# ~/.bashrc
if [ -f ~/.bash_aliases ]; then
  . ~/.bash_aliases
fi
```

```
# ~/.bash_aliases
alias ll='ls -lah'
alias dc='docker compose'
```

Then source ~/.bashrc or open a new shell.

Remove

```
unalias ll
unalias -a          # nuclear; current shell only
```

Bypass aliases

```
command ls
\ls
/bin/ls
```

Essential when an alias breaks a scripted expectation interactively.

sudo and alias expansion (bash)

```
alias sudo='sudo '
# trailing space makes bash expand the next word as an alias
sudo ll                # may work if ll is alias
```

Better as functions

```
mkcd() { mkdir -p -- "$1" && cd -- "$1"; }
backup() { cp -a -- "$1" "$1.bak.${date +%Y%m%d%H%M%S}"; }
```

Aliases do not take parameters cleanly; functions do.

Safety alias caveats

```
alias rm='rm -i'
# feels safe; non-interactive scripts still use raw rm
# other users/shells unaffected
```

Notes / Pitfalls

- Non-interactive bash does not expand aliases by default (`shopt expand_aliases` can enable, but functions are clearer).
- Quote carefully: `alias e=echo hi` is wrong; use `alias e='echo hi'`.
- Recursive aliasing and shadowing builtins can confuse (`alias cd=...` — avoid unless intentional).
- `zsh`/`fish` have richer abbreviation systems; syntax differs.
- Exporting aliases to child shells is not a thing — source rc files instead.

2026-relevant notes

- Dotfile managers (chezmoi, yadm) commonly template alias collections per machine.
- Prefer shared functions in a small `~/.config/shell/functions` for portability across bash/zsh when possible.
- Tool-provided completions > heavy aliases for complex CLIs (`kubectl`, `docker`).

Related Commands

- Shell functions — parameterized shortcuts
- `type / command -v` — resolution
- `ln -s` — filesystem shortcuts
- `hash` — bash command hashing
- `bind / keymaps` — interactive productivity beyond aliases

Additional Resources

- `help alias`, `help unalias`, `man bash`

install

Overview

`install` copies files with **explicit mode** (and optional owner/group) and can create destination directories. It is preferred in Makefiles, packaging, and deploy scripts over ad-hoc `cp + chmod + mkdir -p` sequences because intent is clear and modes are set in one step.

Despite the name, it is a userland coreutils program — not a package manager.

Syntax

```
install [options] SOURCE DEST
install [options] SOURCE... DIRECTORY
install [options] -t DIRECTORY SOURCE...
install -d [options] DIRECTORY...
```

Common Options

Option	Description
<code>-d, --directory</code>	Create directories (like <code>mkdir -p</code> with mode)
<code>-m MODE, --mode=MODE</code>	Permission mode (default often 755 for files historically — set explicitly)
<code>-o OWNER, -g GROUP</code>	Owner/group (requires privileges)
<code>-t DIR, --target-directory=DIR</code>	Destination directory
<code>-D</code>	Create leading components of DEST path
<code>-b</code>	Backup existing destination
<code>-S SUF</code>	Backup suffix
<code>-v</code>	Verbose
<code>-p</code>	Preserve timestamps (when supported)
<code>-C</code>	Install only if contents differ (compare; GNU)
<code>--strip / -s</code>	Strip symbol tables from binaries (toolchain)

Key Use Cases

1. Install binaries/scripts to `/usr/local/bin`
2. Drop configs with correct mode/owner
3. Create system directory trees with known permissions
4. Reproducible Makefile `make install` steps
5. Install unit files and server config snippets with correct modes

Examples with Explanations

Install a script or binary

```
sudo install -m 755 bin/tool /usr/local/bin/tool
sudo install -m 755 target/release/app /usr/local/bin/app
```

Create directories

```
sudo install -d -m 755 /etc/myapp
sudo install -d -m 750 -o myapp -g myapp /var/lib/myapp
```

Config with ownership

```
sudo install -m 640 -o root -g adm myapp.conf /etc/myapp/myapp.conf
```

Leading path creation (systemd unit)

```
sudo install -D -m 644 myapp.service \
/etc/systemd/system/myapp.service
sudo install -D -m 644 myapp.conf \
/etc/myapp/myapp.conf
```

`-D` creates parent directories of the final DEST file.

Multiple sources into a directory

```
sudo install -m 644 -t /etc/myapp/ conf.d/*.conf
```

Compare before replace (GNU)

```
sudo install -C -m 644 app.conf /etc/myapp/app.conf
```

Avoids unnecessary writes/timestamp bumps when content is identical.

Makefile fragment

```
PREFIX ?= /usr/local
bindir ?= $(PREFIX)/bin

install:
    install -d $(DESTDIR)$(bindir)
    install -m 755 tool $(DESTDIR)$(bindir)/tool
```

Versus cp

```
# verbose multi-step
mkdir -p /opt/app/bin
cp tool /opt/app/bin/tool
chmod 755 /opt/app/bin/tool

# one tool
install -D -m 755 tool /opt/app/bin/tool
```

Notes / Pitfalls

- Default mode if you forget `-m` may not be what you want — **always set `-m`**.
- `-o/-g` need privileges; packaging often installs as root in `DESTDIR` then packages own meta-data.
- `install` is not atomic across filesystems in a fancy way — still a copy into place; for live configs consider `write-temp + mv`.
- Stripping (`-s`) needs strip tools and is for binaries, not scripts.
- SELinux systems may need `restorecon` after install into labeled paths.

2026-relevant notes

- systemd unit installs: `install -D -m 644 unit /etc/systemd/system/` then `systemctl daemon-reload`.
- Prefer `%install/install` in packaging (RPM/deb helpers) with explicit modes for supply-chain clarity.

- User-level installs under `~/.local` pair well with XDG paths and no root.

Related Commands

- `cp` — general copy
- `mkdir -p` — directories only
- `chmod` / `chown` — adjust after the fact
- `rsync` — sync trees
- `install` target in autotools/Meson/CMake — ecosystem convention
- `ln -s` — link instead of copy

Additional Resources

- `man install`

df

Overview

df reports **filesystem disk space usage** for mounted filesystems. First stop for “is the disk full?” before descending into per-directory **du/dust/ncdu**. It answers capacity questions at mount granularity, not “which folder is large?”.

Syntax

```
df [options] [file...]
```

When given a path, reports the filesystem containing that path.

Common Options

Option	Description
-h, --human-readable	Sizes in powers of 1024
-H	Powers of 1000
-T	Print filesystem type
-i	Inode usage instead of blocks
-l	Local filesystems only
-P	POSIX output format (stable columns)
--output=FIELD_LIST	Select columns (GNU)
-t type / -x type	Include / exclude filesystem type
-a	Include pseudo, duplicate, inaccessible

Examples with Explanations

Everyday overview

```
df -h
df -hT
df -h -x tmpfs -x devtmpfs
```

Inodes exhausted?

```
df -ih
```

Millions of tiny files can exhaust inodes while `df -h` still shows free blocks — common on mail spool or container image layers.

Specific path

```
df -h /var
df -h .
df -h /var/lib/docker
```

Shows the mount that holds that path (critical with bind mounts).

Portable scripting

```
df -P / | tail -1
df -P --output=target,pcent /var
```

JSON-ish via findmnt (often clearer)

```
findmnt -D
findmnt -no TARGET,FSTYPE,USE% /
```

Correlate devices

```
df -hT
lsblk -f
findmnt
```

Root reserved space

On ext*, ~5% may be reserved for root. Use% at 100% can still allow root writes; non-root fails earlier.

Understanding output

Field	Meaning
Filesystem	Device or source
Size / Used / Avail	Block capacity
Use%	Used / size (rounding applies)
Mounted on	Mountpoint
Type (-T)	ext4, xfs, btrfs, nfs, ...

Notes / Pitfalls

- tmpfs/overlay/snap noise hides real disks — filter with `-x / -t / duf -only local`.
- NFS free space can reflect server quotas, not your expectation.
- Btrfs/ZFS free space may need native tools for accurate “allocatable” views.
- Deleted-but-open files still consume space until close (`lsof + (deleted)`).
- Don’t parse human `-h` in scripts — use `-P` or `--output bytes` fields.

2026-relevant notes

- Prefer `duf` interactively; keep `df` in scripts and alerts.
- Container hosts: full `/var/lib/containerd` or `docker` roots are common; `df` the right mount.
- Metrics: node exporters scrape the same ideas — alert on both block and inode %.

Related Commands

- `du / dust / ncd` — directory usage
- `duf` — modern `df` UI
- `lsblk / findmnt` — topology and mounts
- `lsof` — open deleted files holding space
- `btrfs filesystem usage / zpool list` — advanced FS free space

Additional Resources

- `man df`

du

Overview

du estimates **file and directory space usage** by walking the tree. Use it after **df** shows a full mount to find which directories consume space. For interactive exploration, prefer **ncdu**/**dust**; for scripts, **du -sb** or **du -sh**.

Syntax

```
du [options] [file...]
```

Common Options

Option	Description
-h	Human-readable
-s	Summarize (total only)
-a	All files, not just dirs
-c	Grand total
-d N, --max-depth=N	Limit depth
-x	One filesystem only
-b / -k / -m	Bytes / KiB / MiB
--apparent-size	Logical size vs allocated blocks
-L	Follow symlinks
--exclude=PATTERN	Skip matching paths
-t SIZE	Threshold (GNU)
--inodes	Count inodes (GNU)
-0	NUL-terminated output

Examples with Explanations

Top-level breakdown

```
du -h -d 1 /var 2>/dev/null | sort -h
du -sh /var/*
```

Summaries

```
du -sh .
du -sh /home/*
du -c -sh /var/log /var/cache
```

One filesystem (skip mounts)

```
du -x -h -d 1 /
```

Avoids descending into `/mnt`, network mounts, etc.

Apparent vs allocated

```
du -sh file.img
du -sh --apparent-size file.img
# sparse files differ a lot
```

Exclude noise

```
du -sh --exclude='.git' --exclude='node_modules' *
```

Sort biggest

```
du -h -d 1 | sort -h
du -h -d 2 /var | sort -h | tail
```

Scripts (bytes)

```
du -sb /var/lib/myapp | awk '{print $1}'
```

Find large files (pair with find)

```
find /var -xdev -type f -size +1G -exec ls -lh {} \;  
du -ah /var 2>/dev/null | sort -h | tail -20
```

Notes / Pitfalls

- Permission denied dirs undercount silently unless stderr watched.
- Hard links: GNU du counts once per tree walk by default when possible.
- Different mounts under a path inflate totals unless `-x`.
- Snapshots (btrfs/zfs) make “usage” subtle — use native tools for pool accounting.
- `sort -h` needs human units from `du -h`.

2026-relevant notes

- Interactive: `dust`, `ncdu`, `dua`. Scripts: stick to `du`.
- Container graph drivers: measure the host path carefully; overlay layers confuse naive `du`.
- Combine `df` (full?) + `du` (where?) + `lsof` (deleted open files).

Related Commands

- `df` — filesystem free space
- `dust` / `ncdu` — interactive usage
- `find -size` — size predicates
- `ls -lh` — single file sizes
- `btrfs filesystem du` — special accounting

Additional Resources

- `man du`

bat

Overview

bat is a modern cat replacement with **syntax highlighting**, git integration, line numbers, and automatic paging. Optional package on most distros (`apt install bat` — binary may be `batcat` on Debian/Ubuntu). Excellent for reading code and configs interactively; keep `cat` for portable scripts.

Syntax

```
bat [options] [file...]
batcat [options] [file...] # Debian/Ubuntu package name collision
```

Common Options

Option	Description
<code>-n, --number</code>	Show line numbers (often default in some themes)
<code>-p, --plain</code>	Plain mode (no decorations; closer to cat)
<code>-A, --show-all</code>	Show non-printable characters
<code>-l, --language</code>	Force language for highlighting
<code>-r, --line-range</code>	Only print a range (N:M)
<code>-H, --highlight-line</code>	Highlight a line
<code>-C, --context</code>	Context lines around highlights (with grep integration)
<code>--style</code>	Configure decorations (<code>auto,full,plain,numbers,...</code>)
<code>--theme</code>	Color theme
<code>-f, --force-colorization</code>	Color even when piping (careful)
<code>--paging=never</code>	Disable pager
<code>-d, --diff</code>	Show git diff modifications
<code>-m, --map-syntax</code>	Map glob → language

Examples with Explanations

View files

```
bat README.md
bat /etc/ssh/sshd_config
bat src/main.rs
```

Debian binary name

```
command -v bat batcat
alias bat=batcat          # common convenience on Ubuntu
batcat -p file.txt
```

Plain / script-friendly

```
bat -p file.txt
bat --paging=never -p file.txt | wc -l
```

Ranges and highlights

```
bat -r 20:40 file.go
bat -H 42 config.yaml
bat -r :50 -H 10 app.py
```

Force language

```
bat -l json mystery.conf
bat -l yaml /etc/netplan/01-netcfg.yaml
```

Multiple files and stdin

```
bat *.md
command | bat -l log
curl -fsSL https://example.com/file.yaml | bat -l yaml
```

Git-aware

```
bat -d src/main.rs           # highlight modifications when in a git repo
```

Themes and style

```
bat --list-themes
bat --theme=TwoDark file.rs
bat --style=numbers,changes file.rs
```

As man / pager helper

```
export MANPAGER="sh -c 'col -bx | bat -l man -p'"
# or use bat's own man integration if configured
```

Compare with cat / less

```
cat file.md                 # raw dump
bat file.md                 # highlight + pager
less -R file.md             # pager only
```

Notes / Pitfalls

- Not installed by default on servers; scripts should not require **bat**.
- Color in pipes can break consumers — default is usually no color when non-tty; use `-p/--paging=never` when unsure.
- Debian renames to **batcat** because of another **bat** package.
- Huge minified JSON/logs: disable paging or use **less/jq** for structure.
- Theme readability depends on terminal truecolor support.

2026-relevant notes

- Pairs well with **eza**, **fd**, **rg**, **delta** as a modern CLI toolkit.
- **batcache** / automatic syntax detection covers most languages; force `-l` for unusual extensions.
- In CI, stick to **cat** for deterministic plain logs.

Related Commands

- `cat` — portable concatenate/print
- `less` — pager
- `highlight` / `pygmentize` — other highlighters
- `delta` — git diff highlighter
- `jq` — structured JSON view
- `man` — manuals (can pipe through `bat`)

Additional Resources

- `man bat` / `bat --help`
- [sharkdp/bat](#)

eza

Overview

`eza` is a modern replacement for `ls` (community successor to `exa`) with colors, git status, icons (optional), tree view, and friendlier defaults. Install separately (`apt install eza`, `cargo install eza`, etc.). Keep knowing classic `ls` for minimal systems and scripts.

Syntax

```
eza [options] [path...]
```

Common Options

Option	Description
<code>-l, --long</code>	Long format
<code>-a, --all</code>	Show hidden (not <code>.</code> / <code>..</code>)
<code>-A</code>	Show hidden including <code>.</code> / <code>..</code> variants per version
<code>-h, --header</code>	Header row in long mode
<code>-g, --group</code>	Show group
<code>-b, --binary</code>	Binary (power-of-two) sizes
<code>-B, --bytes</code>	Exact bytes
<code>--git</code>	Git status column
<code>--git-ignore</code>	Ignore gitignored entries
<code>-T, --tree</code>	Tree view
<code>-L, --level</code>	Tree depth
<code>-s, --sort</code>	Sort field (<code>name</code> , <code>size</code> , <code>modified</code> , ...)
<code>-r, --reverse</code>	Reverse sort
<code>-t, --time / --time-style</code>	Time field / format
<code>-i, --inode</code>	Inodes
<code>-u / -U</code>	User / don't show user
<code>--icons</code>	Icon glyphs (needs font)
<code>-1</code>	One per line
<code>-d</code>	List directories as dirs, not contents
<code>-R</code>	Recurse without tree graphics

Flags evolve quickly — `eza --help` is authoritative for your version.

Examples with Explanations

Everyday aliases

```
eza
eza -la
eza -lah --git
eza -lah --group --header
```

Suggested aliases:

```
alias ls='eza'
alias ll='eza -lah --git'
alias lt='eza -T -L 2'
```

Sorting

```
eza -l --sort=modified
eza -l --sort=size -r
eza -l --sort=ext
```

Tree view

```
eza -T -L 2
eza -T -L 3 --git-ignore
eza -T -L 2 -a
```

Git-aware listing

```
eza -l --git
eza -l --git --git-ignore
```

Shows modified/untracked markers when inside a repository.

Headers and metadata

```
eza -lbh --header --group
eza -l --inode --links
```

Compare with ls / tree

```
ls -lah
eza -lah --git
tree -L 2
eza -T -L 2
```

Scripts: prefer classic ls

```
# portable scripts
ls -1
# interactive human shell
eza -lah --git
```

Notes / Pitfalls

- Not installed everywhere; don't hard-depend in production automation.
- Icons need a Nerd Font / patched font or they look like tofu boxes.
- Color/git features add cost on huge directories and network FS — fall back to `ls -U/find` when needed.
- Option names differ from `ls`; muscle memory transfer is incomplete.
- Output is still human-oriented; don't parse it in scripts.

2026-relevant notes

- `eza` is the maintained path for many who used `exa`.
- Fits a modern toolkit with `bat`, `fd`, `rg`, `delta`, `zoxide`.
- Remote SSH to Alpine/BusyBox hosts: expect plain `ls` only.

Related Commands

- `ls` — portable listing
- `tree` — classic tree
- `lsd` — another modern `ls`
- `fd` / `find` — selection
- `stat` — precise metadata
- `git status` — full git detail

Additional Resources

- `eza --help`, `man eza` (if packaged)
- [eza-community/eza](#)

sha256sum

Overview

`sha256sum` computes or verifies **SHA-256** message digests. Operators use it to confirm ISO/downloads, detect bit-rot or accidental corruption, and check published checksums. It is not encryption and does not prove a file is free of malware — only that content matches an expected hash (or another copy).

Related tools: `sha1sum`, `md5sum` (weaker; avoid for new security checks), `b2sum`, `sha512sum`.

Syntax

```
sha256sum [options] [file...]  
sha256sum -c [options] checksums.txt
```

With no files (or `-`), reads stdin. Default output: `<hash> <filename>` (two spaces = text mode).

Common Options

Option	Description
<code>-b, --binary</code>	Binary mode marker (*filename)
<code>-t, --text</code>	Text mode (default on GNU/Linux)
<code>-c, --check</code>	Read checksums from files and verify
<code>--ignore-missing</code>	Skip missing files when checking
<code>--quiet</code>	When checking, only print failures
<code>--status</code>	Silent; exit status only
<code>--strict</code>	Non-zero exit on malformed lines
<code>-w, --warn</code>	Warn about improperly formatted lines
<code>--tag</code>	BSD-style output

Examples with Explanations

Hash a file

```
sha256sum ubuntu-24.04.1-live-server-amd64.iso
# e3b0c442... ubuntu-24.04.1-live-server-amd64.iso
```

Compare the printed digest to the publisher's checksum page (preferably over HTTPS / signed).

Hash several files

```
sha256sum *.iso
sha256sum file1 file2 | tee SHA256SUMS
```

Stdin

```
echo -n 'hello' | sha256sum
cat archive.tar | sha256sum
```

Note: `echo` adds a newline unless `-n`; that changes the hash.

Write and verify a checksum file

```
sha256sum image.iso config.yaml > SHA256SUMS
sha256sum -c SHA256SUMS
# image.iso: OK
# config.yaml: OK
```

Keep `SHA256SUMS` next to the files; run `-c` after transfer or storage.

Quiet / script-friendly check

```
sha256sum -c --status SHA256SUMS && echo all-ok || echo FAIL
sha256sum -c --quiet SHA256SUMS
```

`--status` is ideal for CI: no stdout noise, rely on exit code.

Verify published ISO pattern

```
# After downloading both the ISO and SHA256SUMS (or .sha256) from a trusted source:
sha256sum -c SHA256SUMS 2>&1 | grep iso
# or single line:
echo 'EXPECTEDHASH ubuntu.iso' | sha256sum -c -
```

Always obtain the expected hash from a **trustworthy** channel (vendor site, signed list), not an untrusted mirror alone.

Directory of artifacts

```
find dist -type f -print0 | sort -z | xargs -0 sha256sum > dist/SHA256SUMS
(cd dist && sha256sum -c SHA256SUMS)
```

Stable sort keeps the checksum file reproducible across machines.

Compare two local copies without a list

```
sha256sum a/big.img b/big.img
# same hash → same content (collision risk is negligible for accidental errors)
```

For byte-identical compare of two paths, `cmp` is enough; hashes shine when one side is remote or listed.

Notes

- On GNU/Linux the “text vs binary” distinction rarely changes results for normal files; Windows tools may differ in line-ending handling.
- Renaming a file does not change its hash; only content does.
- Truncation/corruption almost always changes the digest — a matching hash is strong integrity evidence against random errors.
- SHA-256 is a one-way fingerprint, not a MAC. For authenticity against an attacker, use **signed** checksums (GPG/cosign) or TLS from a trusted origin.
- Huge files: hashing is CPU + full sequential read; expect time proportional to size.

Related Commands

- `md5sum` / `sha1sum` / `sha512sum` / `b2sum` — other digests
- `cmp` — direct byte compare of two files
- `diff` — line-oriented text compare
- `openssl dgst -sha256` — OpenSSL digest interface
- `gpg --verify` — signature verification for published artifacts

Additional Resources

- `man sha256sum`
- [Ubuntu installation checksum documentation \(per-release\)](#)

base64

Overview

base64 encodes binary data to ASCII text, or decodes it back. Common uses: paste small secrets into YAML/JSON, embed short binaries in scripts, and read tokens from cloud/K8s manifests. It is **encoding, not encryption** — anyone can decode it.

Syntax

```
base64 [options] [file]
base64 --decode [options] [file]
```

Default reads stdin or a single file and writes encoded text to stdout (GNU coreutils).

Common Options

Option	Description
-d, -D, --decode	Decode
-w COLS, --wrap=COLS	Wrap encoded lines at COLS (0 = no wrap; GNU default often 76)
-i, --ignore-garbage	When decoding, ignore non-alphabet characters
(file)	Input file; omit or - for stdin

macOS base64 flags differ slightly (-D vs -d); this page targets GNU/Ubuntu.

Examples with Explanations

Encode a file

```
base64 secret.key
base64 -w0 secret.key > secret.b64      # single line (scripts/YAML)
```

-w0 avoids line breaks that break some config parsers.

Decode to a file

```
base64 -d secret.b64 > secret.key
base64 --decode -i pasted.txt > out.bin
```

Encode/decode stdin pipelines

```
echo -n 'hello' | base64
echo 'aGVsbG8=' | base64 -d
printf 'hello' | base64 -w0
```

Prefer `printf` when you must control trailing newlines precisely.

Small binary in a shell variable (careful)

```
B64=$(base64 -w0 icon.png)
echo "$B64" | base64 -d > icon-copy.png
cmp icon.png icon-copy.png && echo identical
```

Large blobs belong in files, not shell variables.

Kubernetes-style secret (illustrative)

```
echo -n 's3cr3t' | base64 -w0
# put result in Secret data: fields - still not encryption at rest without cluster config
```

Anyone with API read access can base64-decode Secret data unless encryption-at-rest is configured.

PEM / cert peek

```
# Intermediate: binary DER to base64 lines is what PEM body is
base64 -w64 cert.der
openssl x509 -in cert.pem -noout -subject # prefer openssl for real cert work
```

Ignore whitespace garbage when decoding

```
base64 -d -i email-attachment.b64 > file.bin
```

Helpful when copy-paste introduced spaces or headers.

Notes

- Encoded size $\frac{4}{3}$ of input (plus newlines if wrapping).
- Base64 is reversible and **not** a confidentiality control.
- Wrong alphabet/variants (URL-safe base64) may need `tr -- '+/' '-_'` transforms or other tools.
- Always use `-w0` for single-line config values unless the format requires classic MIME wrap.
- Decoding invalid input may produce partial garbage — verify with length checks or hashes when it matters.

Related Commands

- `xxd / hexdump` — hex views of binary
- `openssl base64` — alternate CLI
- `sha256sum` — integrity after transfer
- `jq -r` — often used with JSON that embeds base64 fields

Additional Resources

- `man base64`

cmp

Overview

`cmp` compares two files **byte by byte**. It reports the first difference (byte number and line number) or stays silent if they match. Use it to verify copies, compare binaries, or confirm a write succeeded. Prefer `diff` for human-readable text changes; prefer `sha256sum` when comparing against a published digest or many files via a list.

Syntax

```
cmp [options] file1 file2
```

Either file may be `-` for stdin (not both in all cases — see man page).

Common Options

Option	Description
<code>-l, --verbose</code>	List all differing byte positions and values
<code>-s, --quiet, --silent</code>	No output; exit status only
<code>-n BYTES, --bytes=BYTES</code>	Compare at most BYTES
<code>-i BYTES,</code> <code>--ignore-initial=BYTES</code>	Skip first BYTES of both files
<code>-i BYTES1:BYTES2</code>	Skip different prefixes per file
<code>-b, --print-bytes</code>	Show differing bytes (with <code>-l</code>)

Examples with Explanations

Basic equal / not equal

```
cmp a.bin b.bin && echo same || echo differ
cp a.bin a.copy
cmp a.bin a.copy && echo copy-ok
```

Silent success + exit 0 means identical through EOF of the shorter comparison path.

Where they first differ

```
cmp report-v1.dat report-v2.dat
# report-v1.dat report-v2.dat differ: byte 1024, line 1
```

Byte index is 1-based in GNU cmp messages. Line number counts newlines seen — less meaningful for pure binary.

Quiet for scripts

```
if cmp -s "$src" "$dst"; then
    echo 'unchanged'
else
    echo 'needs update'
fi
```

List every difference (small files)

```
cmp -l file1 file2 | head
# byte_num  octal_val1  octal_val2
```

Noisy for large divergent files — stop with `head` or use `diff/sha256sum` instead.

Compare after skipping headers

```
cmp -i 512 disk.img partition.img
cmp -i 0:2048 full.img payload.bin
```

Useful when one side has a preamble (MBR, archive header) the other lacks.

Limit length

```
cmp -n 1M image.iso /dev/sdX
```

Compares only the first 1 MiB — e.g. quick check that a written ISO prefix matches without reading the whole medium.

Against a device (careful read-only)

```
sudo cmp -n "$(stat -c%s ubuntu.iso)" ubuntu.iso /dev/sdX && echo usb-matches-iso-prefix
```

Read-only verification after `dd`. Still confirm device identity with `lsblk` first.

Notes

- Exit status: 0 same, 1 different, 2 error (missing file, etc.).
- `cmp` stops at the first difference unless `-l`.
- For large “are these the same?” checks over the network, hashing or `rsync -nc` may be more practical.
- Text normalization (CRLF vs LF) is **not** ignored — bytes must match.
- Sparse files and special files follow normal read semantics; comparing devices needs privileges and caution.

Related Commands

- `diff` — line-oriented text
- `sha256sum` — digest compare / published checksums
- `md5sum` — weaker legacy digest
- `comm` — compare sorted line sets
- `rsync -nc` — dry-run content check via `rsync`

Additional Resources

- `man cmp`

getfacl / setfacl

Overview

POSIX **ACLs** extend the classic owner/group/other mode bits with named user and group entries. On Ubuntu, **getfacl** displays ACLs; **setfacl** changes them. Use ACLs when several users/groups need different rights on shared directories without resorting to **777** or endless group reshuffles. Classic **chmod** bits remain the base; ACL masks interact with them — always verify with **getfacl** after changes.

Package: **acl** (**sudo apt install acl** if missing). Filesystem must be mounted with ACL support (default for ext4/xfs on modern Ubuntu).

Syntax

```
getfacl [options] file...
setfacl [options] file...
setfacl -b|-k file...          # remove ACLs / default ACLs
```

Common Options

getfacl

Option	Description
-a, --access	Access ACL only
-d, --default	Default ACL only (directories)
-c, --omit-header	Skip file name header
-R, --recursive	Recurse
-p, --absolute-names	Don't strip leading /
--all-effective	Comment effective rights (mask)

Option	Description
--------	-------------

setfacl

Option	Description
<code>-m, --modify=acl</code>	Modify ACL entries
<code>-M file</code>	Modify from ACL file
<code>-x, --remove=acl</code>	Remove entries
<code>-b, --remove-all</code>	Remove all extended ACL entries
<code>-k, --remove-default</code>	Remove default ACL
<code>-R, --recursive</code>	Recurse
<code>-d, --default</code>	Apply to default ACL (dirs)
<code>--set=acl</code>	Replace entire ACL
<code>-n, --no-mask</code>	Don't recalculate mask (advanced)
<code>--test</code>	Dry-run: show resulting ACL

ACL entry forms: `u:user:rwX`, `g:group:rX`, `m::rX`, `o::r`, `u::rwX` (owner), `g::rX` (owning group).

Safety

- Recursive `setfacl -R` can lock out services if you omit needed identities — test on a small tree first (`--test` / non-prod).
- The **mask** entry caps effective rights for named users/groups; surprising “I granted `rwX` but it's only `r-x`” is usually the mask.
- Removing ACLs (`-b`) falls back to ordinary mode bits — confirm those bits with `ls -l` afterward.
- Network shares (NFS/SMB) and containers may map IDs differently; ACL usernames must resolve on the host that enforces them.

Examples with Explanations

View ACLs

```
getfacl /srv/shared
getfacl -p /srv/shared/file.txt
ls -l /srv/shared
# files with ACLs show a trailing '+' on the mode, e.g. -rw-rwxr--+
```

Grant a user access

```
sudo setfacl -m u:alice:rwX /srv/shared
sudo setfacl -m u:bob:r-x /srv/shared
getfacl /srv/shared
```

Grant a group and adjust mask

```
sudo setfacl -m g:devs:rwX /srv/shared
getfacl /srv/shared
# check "effective:" comments if mask limits rights
```

Default ACL on a directory (inherit for new files)

```
sudo setfacl -d -m u:alice:rwX /srv/shared
sudo setfacl -d -m g:devs:rwX /srv/shared
sudo setfacl -d -m o::--- /srv/shared
touch /srv/shared/newfile
getfacl /srv/shared/newfile
```

Default ACLs apply to **new** children; existing files need a separate recursive pass if required.

Recursive grant

```
sudo setfacl -R -m g:devs:rwX /srv/shared
sudo setfacl -R -d -m g:devs:rwX /srv/shared
```

Capital X in ACL text means execute only if the file is a directory or already executable — same idea as chmod's X.

Remove entries / clear ACLs

```
sudo setfacl -x u:bob /srv/shared
sudo setfacl -b /srv/shared/file.txt      # strip extended ACL
sudo setfacl -k /srv/shared              # strip default ACL only
getfacl /srv/shared
```


Copy ACL from one file to another

```
getfacl file1 | sudo setfacl --set-file=- file2
# or
sudo getfacl --access file1 | sudo setfacl --set-file=- file2
```

Dry-run

```
setfacl --test -m u:alice:rw /srv/shared/file.txt
```

Shows the ACL that would be applied without writing it.

Backup and restore ACL tree

```
getfacl -R /srv/shared > shared-acls.backup
# later:
sudo setfacl --restore=shared-acls.backup
```

Paths in the backup must still make sense relative to how `getfacl` recorded them.

Notes

- Install: `sudo apt install acl`.
- `ls -l + flag` means “has ACL (or other security xattr)” — inspect with `getfacl`.
- `umask` and default ACLs both affect new file modes; defaults win for ACL-aware creates when a default ACL exists.
- `chmod` can change the ACL mask as a side effect when ACLs are present — re-check with `getfacl` after `chmod`.
- Not every backup tool preserves ACLs; use `getfacl --restore`, `rsync -A`, or `tar --acls` consciously.

Related Commands

- `chmod` / `chown` — base mode and ownership
- `ls -l` — trailing + hint
- `namei -l` — path component permissions
- `rsync -A` — copy ACLs
- `chattr` — ext file attributes (different mechanism)

Additional Resources

- `man getfacl`, `man setfacl`, `man acl`
- `man 5 acl`

chattr / lsattr

Overview

chattr sets **ext** file attributes (immutable, append-only, no-dump, etc.); **lsattr** lists them. These are separate from mode bits (**chmod**) and ACLs (**setfacl**). Classic operator uses: make a file immutable against accidental deletion, mark logs append-only, or exclude paths from **dump**. Attributes are filesystem-specific — full support is strongest on **ext2/3/4**; other FS may ignore or partially support flags.

Syntax

```
lsattr [options] [file...]  
chattr [options] mode file...
```

mode is += followed by attribute letters, e.g. +i, -a, =i.

Common Options

lsattr

Option	Description
-a	Include dotfiles
-d	List directories themselves, not contents
-R	Recursive
-v	Version/generation if available
-l	Long option names for flags (newer e2fsprogs)

chattr

Option	Description
+X / -X / =X	Add / remove / set exactly attribute X
-R	Recursive
-V	Verbose
-f	Suppress most errors

Option	Description
<code>-v version</code>	Set file version/generation

Important attribute letters

Flag	Meaning (typical)
<code>i</code>	Immutable — cannot modify, delete, rename, link (root must clear first)
<code>a</code>	Append-only — open only in append mode for writes
<code>A</code>	No atime updates
<code>c</code>	Compressed (FS-dependent; often no-op on plain ext4)
<code>d</code>	No dump
<code>e</code>	Extent format (usually set by FS; don't clear casually)
<code>j</code>	Data journaling (ext3/4 specifics)
<code>s</code>	Secure deletion (often not honored as expected)
<code>S</code>	Synchronous updates
<code>u</code>	Undeletable (legacy; rarely useful)

Most operators only need `i` and `a`.

Safety

- `chattr +i` as root can lock out application writes until you remember to `-i` — document it.
- Recursive `+i` on a tree can break package upgrades and log rotation.
- Attributes are **not a substitute for backups or proper permissions**.
- On non-ext filesystems, commands may succeed partially or reject flags — verify with `lsattr` and test writes.
- Immutable system files can block `apt/dpkg` — clear before upgrades.

Examples with Explanations

List attributes

```
lsattr file.txt
lsattr -d /etc/hostname
lsattr -R /var/log/myapp 2>/dev/null | head
```

A line like `----i-----e----- file` means immutable (`i`) plus extent (`e`).

Make a file immutable

```
sudo chattr +i /etc/myapp/critical.conf
lsattr /etc/myapp/critical.conf
# even root cannot overwrite until:
sudo chattr -i /etc/myapp/critical.conf
```

Useful against accidental `rm`/overwrite; not a strong security boundary against a malicious root.

Append-only log

```
sudo chattr +a /var/log/myapp/audit.log
# app can append; cannot truncate/overwrite in place
# to rotate: clear a, rotate, restore a
sudo chattr -a /var/log/myapp/audit.log
```

Coordinate with `logrotate` (`endscript` hooks) if you use `+a`.

Clear / set exactly

```
sudo chattr -i -a file
sudo chattr = file          # clear all settable attrs (careful)
```

Protect a directory entry (directory itself)

```
sudo chattr +i /important-dir
# prevents adding/removing entries in some cases; know the semantics before using
sudo chattr -i /important-dir
```

Test on non-production paths — immutable directories surprise people.

No-dump flag

```
sudo chattr +d /var/cache/bulk
lsattr -d /var/cache/bulk
```

Hints to `dump(8)` to skip; modern backup tools may ignore this flag unless coded for it.

Combine with permissions

```
sudo chown root:root /etc/myapp/critical.conf
sudo chmod 644 /etc/myapp/critical.conf
sudo chattr +i /etc/myapp/critical.conf
```

Mode bits control who can read; `i` resists modification/deletion.

Notes

- Requires appropriate privileges (usually root) for security-sensitive flags like `i` and `a`.
- `e` (extents) is normal on modern ext4 files — don't treat it as a problem.
- XFS has `chattr/lsattr` compatibility for some flags but also its own `xfs_io/xfs_admin` tooling.
- Backups: confirm your backup agent preserves attributes if you rely on them after restore.
- Containers: attributes on bind-mounted host ext volumes work; the container root may still need `CAP_LINUX_IMMUTABLE`-related capabilities for `+i`.

Related Commands

- `chmod` / `chown` — mode and ownership
- `getfacl` / `setfacl` — ACLs
- `ls -l` — basic metadata
- `stat` — inode details
- `chattr` man page attribute list for full alphabet

Additional Resources

- `man chattr`, `man lsattr`
- `man 1 chattr` attribute table

xxd

Overview

xxd creates a **hex dump** of a file (or stdin) and can reverse a dump back to binary. Operators use it to inspect magic numbers, protocol payloads, firmware blobs, and corrupted headers. It ships with Vim on many Ubuntu installs (`vim-common`). Alternatives: `hexdump`, `od`, `hexyl` (nicer TUI).

Syntax

```
xxd [options] [file]
xxd -r [options] [file]
```

Default: hex + ASCII side panel. No file means stdin.

Common Options

Option	Description
<code>-p</code> , <code>-ps</code> , <code>-postscript</code> , <code>-plain</code>	Continuous plain hex (no addresses/ASCII)
<code>-i</code>	C include-style array
<code>-r</code> , <code>-revert</code>	Reverse: hex dump → binary
<code>-r -p</code>	Reverse plain hex
<code>-g N</code> , <code>-groupsize N</code>	Bytes per group (default 2)
<code>-c N</code> , <code>-cols N</code>	Bytes per line
<code>-l LEN</code> , <code>-len LEN</code>	Stop after LEN bytes
<code>-s OFF</code> , <code>-seek OFF</code>	Start at offset (supports +/- relative forms)
<code>-u</code>	Uppercase hex
<code>-b</code>	Bits (binary) dump
<code>-e</code>	Little-endian style grouping (when supported)
<code>-a</code>	Autoskip nul lines (like *)

Examples with Explanations

Basic dump

```
xxd /bin/ls | head
xxd -l 64 file.bin
```

First lines show file offset, hex bytes, and ASCII.

Limit and offset

```
xxd -s 0x100 -l 32 firmware.bin
xxd -s -16 file.bin           # last 16 bytes (GNU/common form)
```

Jump to headers, ELF sections, or trailing magic without dumping the whole file.

Plain hex (scripts / compare)

```
xxd -p -l 16 file.bin
# 7f454c4602010100...
xxd -p file.bin | tr -d '\n' | head -c 32; echo
```

C array for embedding

```
xxd -i blob.bin > blob.h
# unsigned char blob_bin[] = { 0x... };
```

Handy for tiny firmware embeds in C/Go build pipelines (prefer proper resource packing for large assets).

Reverse plain hex to binary

```
echo '68656c6c6f' | xxd -r -p > hello.bin
xxd -p hello.bin
# 68656c6c6f
```

Also works with classic xxd output via `xxd -r`.

Edit a few bytes (workflow)

```
xxd -l 256 config.bin > config.hex
# edit carefully in editor
xxd -r config.hex > config-new.bin
cmp -l config.bin config-new.bin | head
```

For serious binary patching use proper tools; `xxd` is fine for small fixes.

Inspect file magic

```
xxd -l 4 photo.jpg
# ... ff d8 ff e0 ...  JPEG SOI
xxd -l 8 /bin/ls
# ... 7f 45 4c 46 ...  ELF
file photo.jpg /bin/ls
```

Pair with `file` for classification; `xxd` shows the raw bytes.

Grouping and columns for readability

```
xxd -g 1 -c 16 packet.bin | head
xxd -g 4 -c 16 -u header.bin | head
```

Bits view

```
xxd -b -l 8 flags.bin
```

Shows bit patterns — occasional use for flag bytes.

Notes

- Large dumps flood the terminal — always prefer `-l / head` for first look.
- Plain hex reverse (`-r -p`) ignores whitespace; classic reverse expects `xxd`'s address format.
- Package: often `vim-common` on Ubuntu (`dpkg -S $(which xxd)`).
- Offsets in the left column are hexadecimal by default.
- Not a disassembler — use `objdump/llvm-objdump` for instructions.

Related Commands

- `file` — classify content
- `strings` — printable spans
- `od` / `hexdump` — other dump formats
- `cmp` — compare binaries
- `sha256sum` — integrity after edits
- `readelf` / `objdump` — ELF structure

Additional Resources

- `man xxd`

strings

Overview

strings extracts **printable character sequences** from binary files. Operators use it to skim unknown binaries for version strings, URLs, error messages, file paths, and config keys without a full reverse-engineering setup. It is a triage tool — noisy, incomplete, and easy to miss UTF-16 or compressed data.

Syntax

```
strings [options] file...
```

Reads each file and prints sequences of printable characters longer than a minimum length (default often 4).

Common Options

Option	Description
<code>-n N, --bytes=N</code>	Minimum sequence length (default 4)
<code>-a, --all</code>	Scan entire file (not only initialized data sections)
<code>-t o d x</code>	Print offset in octal/decimal/hex before each string
<code>-e enc</code>	Encoding: <code>s</code> (7-bit), <code>S</code> (8-bit), <code>b/l</code> (16-bit big/little), <code>B/L</code> (32-bit)
<code>-w, --include-all-whitespace</code>	Include wider whitespace in sequences
<code>-f, --print-file-name</code>	Prefix each line with filename
<code>-d</code>	Only data sections (object files; opposite of scanning all on some systems)

GNU binutils **strings** is the Ubuntu default (**binutils** package).

Examples with Explanations

Basic scan

```
strings /usr/local/bin/app | less
strings /usr/local/bin/app | grep -i version
```

Longer strings only (less noise)

```
strings -n 10 mystery.bin | head -50
strings -n 8 firmware.img | grep -E 'https?://|\.so|error'
```

Raising `-n` drops short garbage sequences.

Offsets for later xxd

```
strings -t x -n 8 blob | head
# 00004a0 some_version_string
xxd -s 0x4a0 -l 64 blob
```

Hex offsets pair well with `xxd -s`.

UTF-16 little-endian (Windows-ish binaries)

```
strings -e l -n 6 app.exe | head
strings -el -n 6 app.exe | grep -i http
```

Default 7/8-bit scan misses many wide-char strings.

Multiple files

```
strings -f -n 8 /lib/x86_64-linux-gnu/*.so.6 2>/dev/null | grep -i 'OpenSSL\|GNU'
```

`-f` labels which file each hit came from.

Full file scan

```
strings -a -n 8 packed.bin | less
```

Some binaries hide strings outside default sections; `-a` scans everything.

Pipe from another command

```
dd if=/dev/sdX bs=1M count=2 status=none | strings -n 8 | head
curl -fsSL https://example.com/file | strings -n 6 | head
```

Useful for quick content smell tests on images or downloads.

Hunt secrets carefully

```
strings -n 12 config.dump | grep -iE 'password|token|secret|api_key'
```

May surface accidental leaks in binaries or core dumps — handle output as sensitive.

Notes

- Compressed or encrypted regions will not yield meaningful plaintext.
- Minimum length trade-off: low `-n` → noise; high `-n` → missed short tokens.
- Not a substitute for proper malware analysis sandboxes.
- Output can be huge; always filter with `grep/head/less`.
- License strings and compiler markers often identify build toolchain and versions.

Related Commands

- `file` — type classification
- `xxd` / `hexdump` — raw bytes
- `readelf` / `objdump` — structured ELF
- `grep -a` — treat binary as text for patterns
- `binwalk` — firmware carving (separate install)
- `nm` / `objdump -T` — symbol tables when not stripped

Additional Resources

- `man strings`

namei

Overview

`namei` walks a path component by component, showing whether each element is a file, directory, or symlink — and optionally the permission checks a process would face. Excellent for debugging “permission denied” on deep paths and symlink loops.

Package: usually `util-linux` (preinstalled on Ubuntu).

Syntax

```
namei [options] pathname...
```

Common Options

Option	Description
<code>-l</code>	Long listing (mode, owner, group)
<code>-m</code>	Mode bits only style (see man)
<code>-o</code>	Show owner
<code>-v</code>	Vertical layout
<code>-x</code>	Show mount point crossings
<code>-n</code>	Don't follow symlinks (show link itself)

Examples with Explanations

Diagnose a path

```
namei -l /var/www/html/index.html
```

Reveals which directory lacks `x` for your user or which symlink target is wrong.

Follow ownership along the way

```
namei -l /home/alice/.ssh/authorized_keys
```

Classic SSH “modes are too open” / wrong ownership investigations.

Mount boundaries

```
namei -x /mnt/data/subdir/file
```

Shows when a path crosses into another mount.

Notes & Pitfalls

- Interprets the path as the **calling user** would for access checks when using modes.
- Symlink cycles are reported instead of hanging forever.
- Still verify with `ls -ld` on each component for ACLs (`getfacl`).

Related Commands

- `ls -ld` — single component metadata
- `realpath` / `readlink -f` — resolve final path
- `stat` — inode details
- `getfacl` — ACL beyond mode bits

Additional Resources

- `man namei`

umask

Overview

`umask` sets the **file mode creation mask** for the current shell: bits that are turned *off* when new files and directories are created. It is a shell built-in (and a system call); this page covers the interactive/operator usage. Default Ubuntu login shells often use 0002 or 0022.

Syntax

```
umask [-p] [-S] [mode]
```

Common Options

Option	Description
<i>(none)</i>	Print current mask (octal)
-S	Symbolic print (u=rwx,g=rx,o=rx)
-p	Print as a reusable <code>umask ...</code> command
mode	Set mask (octal like 027 or symbolic)

Key Use Cases

1. Tighten default permissions for shared servers
2. Understand why new files are 644 vs 664
3. Script sections that create secrets with stricter modes

Examples with Explanations

Show current mask

```
umask
umask -S
```

How mask applies

```
umask 022
touch /tmp/a; mkdir /tmp/d
ls -l /tmp/a /tmp/d
# file 666 & ~022 → 644; dir 777 & ~022 → 755
```

Base modes before umask: files 666, directories 777 (exec bits matter for dirs).

Stricter group isolation

```
umask 027
# new files 640, dirs 750 - others get nothing
```

Secrets in a script

```
old=$(umask)
umask 077
install -m 600 /dev/null "$HOME/.config/myapp/token"
umask "$old"
```

Prefer `install -m` / explicit `chmod` for critical files; `umask` is a safety net.

Persist for login shells

```
# ~/.profile or /etc/profile.d/umask.sh
umask 027
```

systemd user services and cron jobs may **not** read your interactive profile — set `umask` there too if needed.

Notes & Pitfalls

- `umask` does not change existing files (`chmod` does).
- ACLs default masks on directories can further alter effective rights.
- Numeric `umask` is **not** the resulting mode; it is the bits stripped from the base.
- BusyBox/dash vs bash symbolic forms can differ slightly.

Related Commands

- `chmod` / `chown` — explicit modes/ownership
- `install` — create files with mode in one step
- `getfacl` / `setfacl` — ACL defaults for shared dirs
- `stat` — inspect resulting mode

Additional Resources

- `help umask` (bash)
- `man 2 umask`

Archives and Compression

Intro

Intro

Pack, compress, and inspect archives for backup, distribution, and log history. Prefer **tar** with modern compressors (**zstd**, **xz**) for Linux trees; use **zip** when exchanging with non-Unix tools.

Commands in this part

Command	Role
tar	tar packs multiple files into one archive (and extracts them).
gzip	gzip compresses or decompresses files using the DEFLATE algorithm (.gz).
xz	xz compresses and decompresses files in the XZ format (LZMA2), typically yielding better ratios than gzip / bzip2 at...
bzip2	bzip2 compresses files with the Burrows–Wheeler algorithm (.bz2).
zstd	zstd (Zstandard) is a modern compression tool offering an excellent speed vs ratio curve, levels from very fast to...
zip	zip creates ZIP archives widely used for cross-platform interchange (Windows, macOS, Linux).
unzip	unzip lists, tests, and extracts ZIP archives.
zcat	zcat writes gzip -compressed files to stdout as if they were uncompressed (equivalent to gzip -dc).
zgrep	zgrep runs grep over gzip -compressed files without manual decompression.

Suggested starting points

1. Create/extract trees: **tar** (+ **zstd**/**gzip**/**xz**).
2. Single-file compressors: **gzip**, **xz**, **bzip2**, **zstd**.
3. Interchange with Windows/macOS users: **zip** / **unzip**.

4. Search rotated logs without unpacking: `zgrep`, `zcat`.

Related parts

- Files and paths — copy and verify before archiving
- Text and pipes — process extracted streams
- Networking — `rsync`/`scp` for transfer after packing

Continue with the individual command pages in this part.

tar

Overview

`tar` packs multiple files into one archive (and extracts them). Compression is added via flags or pipes (`gzip`, `xz`, `zstd`). GNU `tar` is the Linux default.

Syntax

```
tar [OPTIONS] -f ARCHIVE [FILES...]
```

Mnemonic ops: `c`reate, `x`tract, `t`list, plus compression letters.

Common Options

Option	Description
<code>-c</code>	Create
<code>-x</code>	Extract
<code>-t</code>	List
<code>-f file</code>	Archive path (required in practice)
<code>-v</code>	Verbose
<code>-z</code>	gzip
<code>-J</code>	xz
<code>-I zstd / --zstd</code>	zstd (GNU tar 1.31+)
<code>-C dir</code>	Change directory
<code>-p</code>	Preserve permissions (extract)
<code>--exclude=PAT</code>	Exclude glob
<code>--strip-components=n</code>	Drop n path prefix levels on extract
<code>-a</code>	Auto-compress from archive suffix

Key Use Cases

1. Source/release tarballs

2. Backup directories
3. Move trees between hosts (with `ssh`)
4. Inspect packages without unpacking fully

Safety

Extracting as root can overwrite system paths if the archive contains absolute or `../` paths. Prefer extracting as unprivileged user into a clean directory; inspect with `-t` first. Avoid piping untrusted tar to disk as root.

Examples with Explanations

Create gzip archive

```
tar -czvf project.tar.gz project/
```

Create zstd archive

```
tar --zstd -cf project.tar.zst project/  
# or: tar -c project | zstd -T0 -o project.tar.zst
```

List

```
tar -tzvf project.tar.gz | head  
tar -tvvf project.tar.gz | head
```

Extract

```
tar -xzf project.tar.gz  
tar -xzf project.tar.gz -C /tmp/unpack
```

Strip top-level directory

```
tar -xzf project.tar.gz --strip-components=1 -C /opt/app
```


Exclude patterns

```
tar -czf src.tar.gz --exclude='.git' --exclude='node_modules' .
```

Stream over ssh

```
tar -C /var/www -czf - site | ssh backup 'cat > /backups/site.tgz'
ssh backup 'cat /backups/site.tgz' | tar -C /restore -xzf -
```

Append is special

GNU tar can update some archives, but **prefer rebuilding** compressed archives rather than `--append`.

Common Usage Patterns

Quick directory move

```
tar -C /src -cf - mydir | tar -C /dst -xf -
```

Notes & Pitfalls

- Order of options is flexible with GNU tar but put `-f` next to the filename.
- Binary integrity: pair with `sha256sum` for releases.
- Sparse files: `--sparse` when needed.

Related Commands

- `gzip` / `xz` / `zstd` — compressors
- `zip` / `unzip` — PKZip format
- `rsync` — sync without full archive
- `cpio` / `pax` — alternate archivers

Additional Resources

- `man tar`

gzip

Overview

gzip compresses or decompresses files using the DEFLATE algorithm (.gz). It is the most universal compression tool on Unix-like systems — logs, tarballs (tar.gz / .tgz), and HTTP content-encoding all speak gzip. Default behavior **replaces** the input file with file.gz (use -k / --keep to retain the original on modern gzip).

For better speed/ratio tradeoffs on new pipelines, consider **zstd**. For maximum ratio on cold archives, consider **xz**. Keep **gzip** for compatibility.

Syntax

```
gzip [options] [file...]  
gunzip [options] [file...]  
gzcat / zcat [file...]           # decompress to stdout
```

Common Options

Option	Description
-c, --stdout	Write to stdout; keep inputs
-d, --decompress	Decompress (same as gunzip)
-k, --keep	Keep input files
-f, --force	Force overwrite / compress links
-n / -N	Don't save / do save original name & time
-q	Quiet
-r	Recursive directories
-t	Test integrity
-v	Verbose
-1 ... -9	Fastest ... best compression (default 6)
-l	List compressed size / ratio for .gz files

Examples with Explanations

Compress / decompress

```
gzip file.txt           # creates file.txt.gz, removes file.txt
gzip -k file.txt        # keep original
gunzip file.txt.gz
gzip -d file.txt.gz     # same
```

stdout pipelines

```
gzip -c file.txt > file.txt.gz
gunzip -c file.txt.gz > file.txt
tar cf - dir | gzip -9 > dir.tar.gz
gzip -dc dir.tar.gz | tar xf -
```

Level tradeoffs

```
gzip -1 fast.log        # CPU cheap
gzip -9 archive.dat     # smaller, slower
```

Test and list

```
gzip -t file.txt.gz
gzip -l *.gz
```

Recursive logs

```
gzip -r /var/log/old/    # careful: compresses in place
find /var/log -name '*.log' -mtime +7 -exec gzip -9 {} +
```

View without extracting

```
zcat file.txt.gz | less
zgrep 'ERROR' app.log.gz
zless app.log.gz
```

Integrity in scripts

```
if gzip -t backup.gz 2>/dev/null; then
    echo ok
fi
```

Notes / Pitfalls

- Without `-k`, the source file is removed after successful compression.
- Compressing already-compressed data (jpeg, mp4, `.gz`) wastes CPU and may grow size.
- `gzip -r` is easy to over-apply on system directories — scope with `find`.
- Concatenated `.gz` members are valid; some tools only read the first.
- Sparse files and special files: stick to regular files.

2026-relevant notes

- Still the interchange default for tarballs and many APIs.
- Prefer `zstd` for local high-volume logs when all consumers support it.
- `pigz` provides parallel gzip when you need multi-core compression of large streams.

Related Commands

- `gunzip` / `zcat` / `zgrep` / `zless` — decompress / search / page
- `pigz` — parallel gzip
- `xz` / `zstd` / `bzip2` — alternate compressors
- `tar` — archive + compress combo
- `gzip -l` vs `stat` — size inspection

Additional Resources

- `man gzip`

XZ

Overview

xz compresses and decompresses files in the **XZ** format (LZMA2), typically yielding better ratios than gzip/bzip2 at the cost of more CPU and memory — especially at high presets. Common for source tarballs (*.tar.xz), package payloads, and cold archives.

Companion commands: **unxz**, **xzcat**, **xzgrep**.

Syntax

```
xz [options] [file...]  
unxz [options] [file...]  
xzcat [file...]
```

Common Options

Option	Description
-z / -d	Compress (default) / decompress
-c	Write to stdout
-k, --keep	Keep input files
-f	Force
-t	Test integrity
-l	List information about .xz files
-0 ... -9	Compression preset (default 6); higher = smaller/slower/more RAM
-e, --extreme	Try harder within preset (more CPU)
-T N, --threads=N	Multithreaded compression (when built with support)
-v	Verbose
-q	Quiet
--memlimit=SIZE	Cap memory usage

Examples with Explanations

Basic

```
xz file.tar
xz -k file.tar
unxz file.tar.xz
xz -d file.tar.xz
```

Pipelines with tar

```
tar -cJf archive.tar.xz dir/      # tar invokes xz
tar -caf archive.tar.xz dir/      # auto by extension (GNU tar)
xz -dc archive.tar.xz | tar xf -
tar -xJf archive.tar.xz
```

Presets and threads

```
xz -O -T0 fast.dat                # fast, multi-thread (T0 = all cores when supported)
xz -9e cold-archive.img           # max effort; watch RAM
xz -T4 -6 backup.sql
```

Test and list

```
xz -t archive.tar.xz
xz -l *.xz
xz -lv package.tar.xz
```

Keep and stdout

```
xz -kc file > file.xz
xzcat file.xz | less
```

Memory safety on small hosts

```
xz --memlimit=50% -T2 big.img
```

High -9 on a 1 GB VPS can invoke the OOM killer — lower preset or set a memlimit.

Convert from gzip (recompress)

```
gzip -dc old.tar.gz | xz -T0 -c > new.tar.xz
```

Notes / Pitfalls

- Decompression is faster than compression but still heavier than gzip for weak CPUs.
- Multi-threaded mode increases memory roughly with threads × preset needs.
- Not as ubiquitous as .gz on ancient appliances — confirm consumer support.
- Default may delete inputs without -k (like gzip).
- Don't use extreme presets for live log shipping; use zstd/gzip.

2026-relevant notes

- Many distros ship packages and kernel sources as .tar.xz.
- For new internal pipelines, **zstd** often wins on speed with comparable ratio at mid levels; xz remains fine for cold distribution archives.
- **pixz** / threaded xz variants exist for parallelize-friendly workflows.

Related Commands

- **unxz** / **xzcat** / **xzgrep** — decompress helpers
- **gzip** / **zstd** / **bzip2** — alternatives
- **tar -J** / **tar --xz** — archive integration
- **lzma** — legacy LZMA utils (related family)
- **7z** — multi-format archiver

Additional Resources

- `man xz`

bzip2

Overview

bzip2 compresses files with the Burrows–Wheeler algorithm (.bz2). It typically achieves better ratios than gzip but is slower and has largely been superseded by **xz** (better ratio) and **zstd** (better speed/ratio balance) for new work. You will still encounter .bz2 and .tar.bz2 in older archives and some software distributions.

Syntax

```
bzip2 [options] [file...]  
bunzip2 [options] [file...]  
bzipcat [file...]
```

Common Options

Option	Description
-c	Write to stdout
-d	Decompress
-z	Compress (default)
-k	Keep input files
-f	Force
-t	Test integrity
-v	Verbose
-q	Quiet
-s	Small memory mode
-1 ... -9	Block size / effort (100k–900k blocks)

Examples with Explanations

Compress / decompress

```
bzip2 file.txt
bzip2 -k file.txt
bunzip2 file.txt.bz2
bzip2 -d file.txt.bz2
```

Pipelines

```
bzip2 -c file.txt > file.txt.bz2
bzipcat file.txt.bz2 | less
tar cjf archive.tar.bz2 dir/
tar xjf archive.tar.bz2
```

Levels

```
bzip2 -1 quick.dat
bzip2 -9 small.dat
```

Test

```
bzip2 -t file.txt.bz2
```

Search compressed

```
bzgrep 'ERROR' app.log.bz2
bzless app.log.bz2
```

Prefer modern tools when choosing

```
# old style
tar cjf backup.tar.bz2 data/
# modern defaults for new archives
tar --zstd -cf backup.tar.zst data/
# or
tar cJf backup.tar.xz data/
```

Notes / Pitfalls

- Deletes inputs by default without `-k`.
- CPU-heavy relative to `gzip/zstd` for similar tasks.
- Memory scales with block size (`-s` helps constrained systems).
- Less common in new HTTP APIs than `gzip/br/zstd`.
- Parallel variant: `pbzip2` / `lbzip2` if installed.

2026-relevant notes

- Maintain decompression support for legacy artifacts; avoid `bzip2` for new internal pipelines unless required.
- GNU `tar` still supports `-j` / `--bzip2`.
- If an upstream only publishes `.tar.bz2`, trust `tar xjf / bzip2 -d` rather than recompressing without need.

Related Commands

- `bunzip2` / `bzcat` / `bzgrep` / `bzless`
- `gzip` / `xz` / `zstd`
- `tar -j`
- `pbzip2` — parallel `bzip2`

Additional Resources

- `man bzip2`

zstd

Overview

zstd (Zstandard) is a modern compression tool offering an excellent **speed vs ratio** curve, levels from very fast to extremely dense, optional multithreading, and framing with checksums. It is the preferred default for many new local pipelines, package systems, and container layers. Extension: `.zst`.

Syntax

```
zstd [options] [file...]  
zstd -d [options] [file...]  
zstdcat [file...]  
unzstd [file...]
```

Common Options

Option	Description
<code>-d, --decompress</code>	Decompress
<code>-c, --stdout</code>	Write to stdout
<code>-f, --force</code>	Overwrite / compress links
<code>-o file</code>	Output file name
<code>-k / --rm</code>	Keep sources (default keep) / remove sources after success
<code>-t</code>	Test integrity
<code>-T#</code>	Threads (-T0 = all cores)
<code>-#</code>	Level 1–19 (default 3); ultra with <code>--ultra -20...-22</code>
<code>-v</code>	Verbose
<code>-q</code>	Quiet
<code>-l / --list</code>	Show frame info
<code>--train</code>	Build a dictionary (advanced)
<code>-D dict</code>	Compress/decompress with dictionary
<code>--long</code>	Long distance matching mode

Unlike classic gzip defaults, **zstd keeps the original file** unless you pass `--rm`.

Examples with Explanations

Basic

```
zstd file.tar
zstd -d file.tar.zst
zstd -o out.zst file
unzstd file.tar.zst
```

Levels and threads

```
zstd -1 -T0 fast.log
zstd -19 -T0 cold.img
zstd --ultra -22 -T0 deepest.img # very slow/heavy
for l in 1 3 6 9 15 19; do
    zstd -$l -f -o /tmp/t.zst sample.bin && ls -lh /tmp/t.zst
done
```

Pipelines with tar

```
tar -I 'zstd -T0' -cf backup.tar.zst dir/
tar --zstd -cf backup.tar.zst dir/ # GNU tar 1.31+
zstd -dc backup.tar.zst | tar xf -
tar -I zstd -xf backup.tar.zst
```

Integrity

```
zstd -t backup.tar.zst
zstd -l backup.tar.zst
```

Remove source when desired

```
zstd --rm big.img
```

Dictionaries (small similar files)

```
zstd --train -o dict samples/*
zstd -D dict small1 small2
zstd -d -D dict small1.zst
```

Useful for massive numbers of similar tiny JSON/logs.

Notes / Pitfalls

- Confirm consumers understand **.zst** before replacing gzip in public APIs.
- Ultra levels need large memory windows — watch small VMs.
- Default **keeps** inputs; scripts ported from gzip may leave duplicates if they assume deletion.
- **tar --zstd** requires sufficiently new GNU tar; **-I 'zstd -T0'** is explicit and flexible.
- Older enterprise appliances may still demand **.gz**.

2026-relevant notes

- Default choice for many new internal artifacts, caches, and log archives.
- Kernel squashfs, package managers, and container tooling increasingly support zstd natively.
- Compare: **gzip** (universal), **xz** (max ratio, slow), **zstd** (best general default).

Related Commands

- **gzip** / **xz** / **bzip2** — alternatives
- **tar** — archive integration
- **zstdcat** / **zstdgrep** (if packaged)
- **pzstd** — parallel client in some packages
- **lz4** — even faster, lower ratio

Additional Resources

- `man zstd`
- [facebook/zstd](#)

zip

Overview

`zip` creates **ZIP** archives widely used for cross-platform interchange (Windows, macOS, Linux). Unlike `tar` + compressor pipelines, `zip` typically **compresses each member** and stores a central directory. Prefer `tar` + `gzip`/`zstd` for Unix-native backups with permissions/owners; use `zip` when recipients expect `.zip` or you need simple selective compression of mixed files.

Syntax

```
zip [options] archive.zip file...
zip [options] archive.zip -r dir/
```

Common Options

Option	Description
<code>-r</code>	Recurse into directories
<code>-q</code>	Quiet
<code>-v</code>	Verbose
<code>-u</code>	Update existing entries
<code>-m</code>	Move files into zip (delete after)
<code>-e</code>	Encrypt (traditional; weak by modern standards)
<code>-P pass</code>	Password on CLI (visible in process list — avoid)
<code>-x pattern</code>	Exclude patterns
<code>-i pattern</code>	Include patterns
<code>-y</code>	Store symbolic links as links (Unix)
<code>-j</code>	Junk (store) path names — flatten
<code>-0 ... -9</code>	Compression level (0 = store)
<code>-s size</code>	Split archive into segments
<code>-F / -FF</code>	Fix archive

Examples with Explanations

Create archives

```
zip notes.zip *.txt
zip -r project.zip project/
zip -r project.zip project/ -x 'project/node_modules/*' -x '*.git*'
```

Update and list

```
zip -u project.zip project/README.md
zip -sf project.zip                # show files (zipinfo-like)
unzip -l project.zip               # often clearer listing
```

Flatten paths

```
zip -j flat.zip path/to/a.txt path/to/b.txt
```

Store without compression

```
zip -0 -r media.zip photos/        # already-compressed JPEGs
```

Encryption (know the limits)

```
zip -e secret.zip secrets.txt
# traditional zip crypto is weak; prefer 7z AES or age/gpg for real secrets
```

Split archives

```
zip -r -s 100m backup.zip data/
```

Exclude junk

```
zip -r src.zip src/ -x '*/__pycache__/*' -x '*.pyc' -x '*/.DS_Store'
```


Date-stamped backup

```
zip -r "backup-$(date +%Y%m%d).zip" documents/
```

Notes / Pitfalls

- Unix permissions, owners, and special files are a poor fit; **tar** preserves more faithfully.
- Password on the command line (**-P**) leaks via **ps** — interactive **-e** is better but still weak crypto.
- Large file ZIP64 support depends on zip version; very old unzip may fail.
- Default may follow or store links differently than you expect — test with **-y**.
- Windows/macOS clients may mishandle exotic UTF-8 names depending on flags/tools.

2026-relevant notes

- For secure sharing, prefer **age**, **gpg**, or encrypted **7z** over zip's traditional encryption.
- CI artifacts often use zip for multi-OS consumers — keep compression level modest for speed.
- **unzip** / **bsdunzip** / Explorer all need to open your result — avoid exotic extensions.

Related Commands

- **unzip** — extract / list
- **zipinfo** — detailed listing
- **7z** — stronger formats/crypto
- **tar** — Unix-native archives
- **zstd** / **gzip** — stream compressors

Additional Resources

- `man zip`

unzip

Overview

`unzip` lists, tests, and extracts **ZIP** archives. It is the usual counterpart to `zip` on Linux. Options differ from `tar`; path traversal and overwrite behavior deserve care when extracting untrusted archives.

Syntax

```
unzip [options] archive.zip [file...] [-x xfile...] [-d exdir]
```

Common Options

Option	Description
-l	List contents
-v	Verbose listing / version
-t	Test integrity
-o	Overwrite files without prompting
-n	Never overwrite
-d dir	Extract into directory
-j	Junk paths (flatten)
-q	Quiet
-f	Freshen existing files only
-u	Update (freshen + extract new)
-P pass	Password (visible in process list — avoid)
-x pattern	Exclude files
-Z	Zipinfo mode (if compiled)

Examples with Explanations

List and test

```
unzip -l archive.zip
unzip -v archive.zip
unzip -t archive.zip
```

Extract

```
unzip archive.zip
unzip archive.zip -d /tmp/out/
mkdir -p /tmp/out && unzip -d /tmp/out archive.zip
```

Selective extract

```
unzip archive.zip readme.txt
unzip archive.zip 'src/*.c' -d /tmp/src
unzip archive.zip -x '*.o' -d /tmp/out
```

Overwrite policies

```
unzip -o archive.zip      # always overwrite
unzip -n archive.zip      # never overwrite
```

Flatten paths

```
unzip -j archive.zip -d /tmp/flat
```

Pipe / quiet automation

```
unzip -q -o artifact.zip -d "$BUILD_DIR"
```

Untrusted archives — path safety

```
# inspect before extract
unzip -l untrusted.zip | less
# extract into empty sandbox only
mkdir -p /tmp/sandbox && unzip -d /tmp/sandbox untrusted.zip
# watch for entries with .. or absolute paths
unzip -l untrusted.zip | awk '{print $NF}' | grep -E '(^\.\.|^\/|/)'
```

Modern `unzip` versions try to block some unsafe paths; still extract untrusted input in a sandbox.

Password

```
unzip -P 'secret' encrypted.zip      # leaks via ps
unzip encrypted.zip                  # prompts
```

Notes / Pitfalls

- Default extract path is the **current directory** — use `-d` deliberately.
- Filename encoding (CP437 vs UTF-8) can garble non-ASCII names from Windows zips.
- ZIP64 needed for huge archives; old `unzip` fails.
- Don't use `-P` in shared environments.
- `busybox unzip` supports a subset of flags.

2026-relevant notes

- CI jobs should pin `-o/-n` explicitly for non-interactive reliability.
- Prefer artifact scanning before `unzip` on multi-tenant builders.
- For tar-based Linux artifacts, stick to `tar` rather than converting everything to zip.

Related Commands

- `zip` — create archives
- `zipinfo` — detailed metadata
- `7z x` — alternative extractor
- `bsdtar` / `tar` — can read some zips depending on build
- `funzip` — pipe single-member extract

Additional Resources

- `man unzip`

zcat

Overview

zcat writes **gzip-compressed** files to stdout as if they were uncompressed (equivalent to `gzip -dc`). Handy in pipelines without creating temporary files. For `.xz/.zst` use `xzcat/zstdcat` (or `zcat` only where your distro aliases them carefully — do not assume).

Syntax

```
zcat [options] [file...]  
gzip -dc file.gz
```

Examples with Explanations

Read a gzip log

```
zcat /var/log/syslog.1.gz | less  
zcat app.log.gz | grep ERROR | tail
```

Concatenate multiple

```
zcat part-*.gz > combined.log
```

Portable explicit form

```
gzip -dc file.gz | jq .
```

Notes & Pitfalls

- `zcat` on some systems is a symlink to `gzip -dc` and only handles `gzip`.
- Compressed sparse/binary data still expands fully — watch disk when redirecting.
- Prefer `zgrep` when you only need matches.

Related Commands

- `gzip` / `gunzip`
- `zgrep` / `zless`
- `xzcat` / `bzcat` / `zstdcat`
- `tar -xzOf` — extract one member to stdout

Additional Resources

- `man zcat` / `man gzip`

zgrep

Overview

zgrep runs `grep` over **gzip-compressed** files without manual decompression. Useful for rotated logs (*.gz). Variants exist for other compressors (`xzgrep`, `bzgrep`, `zstdgrep`).

Syntax

```
zgrep [grep-options] pattern [file...]
```

Examples with Explanations

Search compressed logs

```
zgrep -H 'error' /var/log/nginx/access.log.*.gz  
zgrep -E 'HTTP/1\.[01]" 5[0-9]{2}' access.log.2.gz
```

Recursive pattern over mixed trees

```
find /var/log -name '*.gz' -print0 | xargs -0 zgrep -H 'OutOfMemory'
```

Count matches

```
zgrep -c 'ERROR' app.log.gz
```


Notes & Pitfalls

- Performance is worse than grepping already-decompressed data — decompress hot logs if you query them constantly.
- Binary matches: use **grep** flags carefully (**-a**, **-I**).
- Confirm whether your **zgrep** shells out to **gzip -dc + grep**.

Related Commands

- **grep** / **rg**
- **zcat** / **zless**
- **journalctl** — live system logs
- **xzgrep** / **zstdgrep**

Additional Resources

- **man zgrep**

System Information

Intro

Intro

Identify the host, kernel, time zone, logged-in users, environment, and high-level hardware inventory. Start here when answering “what machine is this?” and “who is on it?”.

Commands in this part

Command	Role
<code>uname</code>	<code>uname</code> prints system information: kernel name, hostname/nodename, kernel release/version, machine hardware name, and...
<code>hostname</code>	<code>hostname</code> prints or sets the system hostname.
<code>hostnamectl</code>	<code>hostnamectl</code> queries and changes the system hostname and related machine identity fields on systemd systems.
<code>lsb_release</code>	<code>lsb_release</code> prints distribution identification claimed via the Linux Standard Base (LSB) interfaces and...
<code>uptime</code>	<code>uptime</code> shows how long the system has been running, how many users are logged in, and the load average (1, 5, and 15...
<code>date</code>	<code>date</code> prints or sets the system date and time.
<code>timedatectl</code>	<code>timedatectl</code> views and changes the system clock, timezone, RTC settings, and NTP synchronization on systemd systems.
<code>cal</code>	<code>cal</code> displays a calendar in the terminal — current month by default, or a specific month/year.
<code>whoami</code>	<code>whoami</code> prints the effective username of the current user.
<code>id</code>	<code>id</code> prints the real and effective user and group IDs for a process (by default, yours), plus supplementary groups.

Command	Role
<code>who</code>	<code>who</code> shows who is logged in, with terminal, login time, and sometimes host/remote info from <code>utmp/wtmp</code> accounting.
<code>w</code>	<code>w</code> shows who is logged in and what they are doing, plus system uptime and load averages.
<code>last</code>	<code>last</code> shows a history of logins, reboots, and runlevel changes from the <code>wtmp</code> accounting file (typically <code>/var/log/wtmp</code>).
<code>env</code>	<code>env</code> runs a program in a modified environment, or prints the current environment variables when given no command.
<code>free</code>	<code>free</code> summarizes system memory usage: physical RAM and swap.
<code>lscpu</code>	<code>lscpu</code> displays CPU architecture information from <code>sysfs</code> and <code>/proc/cpuinfo</code> : sockets, cores, threads, model name,...
<code>lsmem</code>	The <code>lsmem</code> command displays information about memory ranges and their online/offline status.
<code>lsusb</code>	<code>lsusb</code> lists USB devices connected to the system (buses, IDs, vendor/product strings).
<code>lsmod</code>	<code>lsmod</code> lists loaded kernel modules (name, size, use count, dependents).
<code>hwinfo</code>	The <code>hwinfo</code> command provides comprehensive hardware information.

Suggested starting points

1. Kernel/OS: `uname`, `lsb_release` / `os-release`, `hostnamectl`.
2. Time: `date`, `timedatectl`.
3. Identity: `whoami`, `id`, `who`, `w`, `last`.
4. Resources snapshot: `free`, `lscpu`, `lsmem`, `uptime`.
5. Hardware peek: `lsusb`, `lsmod`, `hwinfo` (deeper inventory is in Hardware).

Related parts

- Hardware — PCI/DMI/disk identity tools

- Processes and jobs — live load after identity checks
- Services and runtime — other systemd `*ctl` tools

Continue with the individual command pages in this part.

uname

Overview

uname prints system information: kernel name, hostname/nodename, kernel release/version, machine hardware name, and sometimes processor/OS fields. Scripts use it to branch on architecture (x86_64 vs aarch64) and kernel release.

Syntax

```
uname [options]
```

Common Options

Option	Description
-a, --all	All information in standard order
-s, --kernel-name	Kernel name (default if no options) e.g. Linux
-n, --nodename	Network node hostname
-r, --kernel-release	Kernel release e.g. 6.8.0-xx
-v, --kernel-version	Kernel build version string
-m, --machine	Machine hardware name e.g. x86_64, aarch64
-p, --processor	Processor type (often unknown on Linux)
-i, --hardware-platform	Hardware platform (often unknown)
-o, --operating-system	OS name e.g. GNU/Linux

Examples with Explanations

Everyday

```
uname
uname -a
uname -r
uname -m
```

Architecture branching

```
case $(uname -m) in
    x86_64) arch=amd64 ;;
    aarch64) arch=arm64 ;;
    armv7l) arch=armhf ;;
    *) echo "unsupported: $(uname -m)" >&2; exit 1 ;;
esac
echo "download package for $arch"
```

Kernel version checks

```
uname -r
# compare carefully - string sort version sort
printf '%s\n' "$(uname -r)" | sort -V
```

vs os-release

```
uname -a
cat /etc/os-release
hostnamectl
```

uname is kernel-focused; distro identity lives in /etc/os-release.

Container note

```
uname -r
# often shows the *host* kernel - containers share it
cat /etc/os-release # container userspace
```

Compact inventory line

```
printf '%s %s %s\n' "$(uname -s)" "$(uname -r)" "$(uname -m)"
```

Notes / Pitfalls

- -p/-i frequently print `unknown` on Linux — use `-m`.
- Don't parse `uname -a` as a stable API; request specific flags.
- Version compares need `sort -V` or proper version libraries.
- Different libc/coreutils may format fields slightly differently.
- WSL and custom kernels still report as Linux with distinctive release strings.

2026-relevant notes

- Multi-arch containers and ARM laptops make `uname -m` essential in install scripts.
- Long-term kernels vs mainline: check `uname -r` against CVE guidance for your distro package, not upstream only.
- Pair with `lscpu` for CPU features (`flags`) when deciding SIMD builds.

Related Commands

- `hostname` / `hostnamectl` — host identity
- `cat /etc/os-release` — distro identity
- `lscpu` — CPU details
- `arch` — often alias-like to `uname -m`
- `hostnamectl` — pretty OS string on systemd

Additional Resources

- `man uname`

hostname

Overview

hostname prints or sets the system hostname. On modern systemd systems, prefer **hostnamectl** for persistent hostname changes and clearer status. The classic **hostname** command remains useful for quick reads and scripts.

Related names: static hostname, pretty hostname, transient hostname (DHCP/cloud).

Syntax

```
hostname [options] [name]
```

Common Options

Option	Description
<i>(none)</i>	Print hostname
name	Set hostname (needs root; often transient without systemd integration)
-f, --fqdn	Fully qualified domain name (best effort)
-s, --short	Short name (before first dot)
-i, --ip-address	Addresses for host (unreliable; prefer ip)
-I	All IP addresses on interfaces (not DNS)
-d	DNS domain name (best effort)
-A	All FQDNs
-y	NIS domain
--help	Help

Behavior of DNS-related flags depends on resolver configuration and **/etc/hosts**.

Examples with Explanations

Read

```
hostname
hostname -s
hostname -f
hostname -I
```

Prefer hostnamectl on systemd

```
hostnamectl
hostnamectl status
sudo hostnamectl set-hostname app-01
```

This updates the persistent hostname properly on systemd hosts.

Temporary classic set (legacy)

```
sudo hostname app-temp
# may not survive reboot without writing config / hostnamectl
```

Scripts

```
h=$(hostname -s)
echo "running on $h"
```

Correlate with DNS / addresses

```
hostname -f
getent hosts "$(hostname -f)"
ip -br addr
```

-i/-f can lie if /etc/hosts and DNS disagree — verify with ip and resolvectl.

Notes / Pitfalls

- Setting via bare `hostname` without `hostnamectl`/config is a common “lost after reboot” mistake.
- Cloud images may regenerate hostnames via cloud-init.
- FQDN detection is best-effort, not a pure DNS authoritative query.
- Containers have their own hostname (pod name, `docker --hostname`).
- Valid hostname rules: prefer short DNS labels (`a-z`, `0-9`, `-`).

2026-relevant notes

- systemd hosts: `hostnamectl` is the supported admin interface.
- Kubernetes sets pod hostnames independently of node hostname.
- Inventory tools (Ansible) often use inventory names that differ from `hostname` — don’t assume equality.

Related Commands

- `hostnamectl` — systemd hostname management
- `resolvectl` / `getent hosts` — name resolution
- `ip` — addresses
- `uname -n` — nodename (usually same as `hostname`)
- `nmcli` — NetworkManager host/domain bits

Additional Resources

- `man hostname`, `man hostnamectl`

hostnamectl

Overview

hostnamectl queries and changes the system hostname and related machine identity fields on **systemd** systems. It is the preferred way to set a **persistent** hostname (vs classic **hostname** alone, which may only change the transient name until reboot).

Syntax

```
hostnamectl [options] [command]
```

Common Commands / Options

Command / option	Description
status	Show current hostnames and OS info (default)
set-hostname NAME	Set hostname
set-hostname NAME --static	Static hostname
set-hostname NAME --transient	Transient only
set-hostname NAME --pretty	Pretty (presentation) name
set-icon-name / set-chassis / set-deployment / set-location	Machine metadata
set-hostname "" --pretty	Clear pretty name
-H host / --machine=	Operate on remote/container via systemd
--no-ask-password	Non-interactive privilege

Examples with Explanations

Status

```
hostnamectl  
hostnamectl status
```

Shows static/transient/pretty hostnames, machine ID icons, chassis, OS pretty name, kernel, architecture.

Set hostname

```
sudo hostnamectl set-hostname app-01  
hostnamectl  
hostname
```

Pretty name

```
sudo hostnamectl set-hostname "App Server 01" --pretty  
sudo hostnamectl set-hostname app-01 --static
```

Transient (temporary)

```
sudo hostnamectl set-hostname temp-probe --transient
```

Chassis / location metadata

```
sudo hostnamectl set-chassis server  
sudo hostnamectl set-location "rack-3-u12"
```

Remote / container

```
hostnamectl -H alice@host.example  
hostnamectl --machine=mycontainer
```

Verify files

```
cat /etc/hostname  
ls -l /etc/machine-id  
hostname
```

Notes / Pitfalls

- Requires systemd; classic SysV-only systems won't have it.
- cloud-init may overwrite hostnames on next boot if configured to manage them.
- DNS is separate: setting hostname does not create DNS records.
- Valid hostname labels: prefer RFC-ish DNS labels for static names; pretty names can be freer.
- Containers may restrict hostname changes.

2026-relevant notes

- Standard admin path on almost all mainstream server distros.
- Inventory tools should read `hostnamectl` status or `/etc/hostname` + `/etc/os-release`.
- Pair with `timedatectl` for complete first-boot identity setup.

Related Commands

- `hostname` — classic get/set
- `timedatectl` — time/timezone
- `localectl` — locale/keymap
- `resolvectl` — DNS
- `cat /etc/os-release` — distro identity

Additional Resources

- `man hostnamectl`

lsb_release

Overview

`lsb_release` prints distribution identification claimed via the Linux Standard Base (LSB) interfaces and distro-specific release files. Still useful for quick “what am I on?” checks, but many modern tools prefer reading `/etc/os-release`, which is the freedesktop standard and always present on current systemd distros.

On some minimal images, `lsb_release` is not installed (`lsb-release` package).

Syntax

```
lsb_release [options]
```

Common Options

Option	Description
<code>-v</code>	LSB modules / version info
<code>-i</code>	Distributor ID
<code>-d</code>	Description
<code>-r</code>	Release number
<code>-c</code>	Codename
<code>-a</code>	All of the above
<code>-s</code>	Short output (no headers)

Examples with Explanations

Everyday

```
lsb_release -a
lsb_release -is
lsb_release -cs
lsb_release -rs
```


Short for scripts

```
distro=$(lsb_release -is 2>/dev/null || true)
codename=$(lsb_release -cs 2>/dev/null || true)
```

Prefer os-release (portable modern)

```
. /etc/os-release
echo "$ID $VERSION_ID $PRETTY_NAME"
# or
grep -E '^(ID|VERSION_ID|PRETTY_NAME)= ' /etc/os-release
```

Compare sources

```
lsb_release -a 2>/dev/null
cat /etc/os-release
hostnamectl | grep -i 'operating system'
```

Install if missing

```
# Debian/Ubuntu
sudo apt install lsb-release
# Fedora
sudo dnf install redhat-lsb-core # may be transitional / optional
```

Often unnecessary if you only need `os-release`.

Notes / Pitfalls

- Not installed everywhere; scripts should fall back to `/etc/os-release`.
- Derived images and containers can customize IDs — trust but verify package availability separately.
- Codename strings differ (`jammy`, `bookworm`, `Trixie`) — don't assume Ubuntu-only.
- LSB compliance fields (`-v`) are historical and less operationally useful today.
- Rolling releases may show unusual version strings.

2026-relevant notes

- **Canonical for automation:** `/etc/os-release` fields `ID`, `ID_LIKE`, `VERSION_ID`.
- Cloud images still include `lsb_release` sometimes for legacy install scripts.
- Prefer feature detection (`command -v, /etc/os-release`) over brittle distro name switches when possible.

Related Commands

- `cat /etc/os-release` — standard identity
- `hostnamectl` — pretty OS string
- `uname` — kernel identity
- `cat /etc/debian_version / redhat-release` — legacy files
- package managers — `apt/dnf` confirm edition

Additional Resources

- `man lsb_release`
- freedesktop os-release specification

uptime

Overview

`uptime` shows how long the system has been running, how many users are logged in, and the **load average** (1, 5, and 15 minutes). It is a quick health snapshot before diving into `top`/`htop`/`vmstat`.

Load average is the average number of runnable **plus** uninterruptible (disk sleep) tasks — not a pure “CPU %” meter.

Syntax

```
uptime [options]
```

Common Options

Option	Description
<code>-p, --pretty</code>	Pretty duration only
<code>-s, --since</code>	System up since timestamp
<code>-h, --help</code>	Help
<code>-V</code>	Version

Examples with Explanations

Default

```
uptime
# 10:15:02 up 21 days,  4:03,  3 users,  load average: 0.08, 0.12, 0.09
```

Pretty / since

```
uptime -p  
uptime -s
```

Interpret load vs CPUs

```
uptime  
nproc  
lscpu | grep '^CPU(s):'  
# rough rule of thumb: load near nproc   busy; much higher may mean saturation
```

Watch over time

```
watch -n1 uptime
```

Scripts

```
load=$(awk '{print $1}' /proc/loadavg)  
echo "1-minute load: $load"  
# /proc/loadavg is the source of truth  
cat /proc/loadavg
```

Correlate with other tools

```
uptime  
who  
w  
top -bn1 | head
```

Containers

```
uptime  
# may reflect host uptime depending on visibility of /proc
```

Understanding load average

Observation	Possible meaning
Low load, high latency	Network/app issues, not CPU
Load nproc, CPU idle% high	I/O wait / uninterruptible tasks
Load runnable tasks	CPU contention
Sudden load spikes	Cron storms, backups, build jobs

Use `vmstat 1`, `iostat -xz 1`, `ps`, `pidstat` for root cause.

Notes / Pitfalls

- Load is not “percent CPU”; a 64-core host can show load 20 and still be fine.
- Suspend-to-RAM / cloud live migration can make “up” time less intuitive.
- User count is from utmp; containers/ssh multiplexing may look odd.
- Don’t alert only on load without CPU, memory, and I/O context.

2026-relevant notes

- For long-term trends use Prometheus node exporter / `sar` rather than one-off `uptime`.
- Latency-sensitive systems care more about p99 and steal time (VMs) than raw load.
- `systemd-analyze` helps boot time; `uptime -s` shows when the current boot began.

Related Commands

- `w` / `who` — who is logged in
- `top` / `htop` / `btm` — live process view
- `vmstat` / `iostat` — run queue vs block I/O
- `nproc` / `lscpu` — CPU count
- `cat /proc/loadavg` — raw values
- `sar` — historical load

Additional Resources

- `man uptime`, `man 5 proc (/proc/loadavg)`

date

Overview

`date` prints or sets the system date and time. Daily use is formatting timestamps for logs, filenames, and scripts. Setting the clock requires privileges and is usually better left to **NTP**/**timedatectl** on modern systems.

Syntax

```
date [options] [+FORMAT]
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]] # set (legacy)
```

Common Options

Option	Description
+FORMAT	Output format string
-u, --utc	UTC
-d STRING, --date=	Display time described by STRING (not set)
-f FILE	Like -d for each line in FILE
-s STRING, --set=	Set time (privileged)
-R	RFC 5322 format
-I [TIMESPEC]	ISO 8601 (<code>date -Iseconds</code>)
-r FILE	Use file mtime

Useful format sequences

Seq	Meaning
%Y-%m-%d	2026-08-04
%H:%M:%S	14:05:09
%F	Same as %Y-%m-%d
%T	Same as %H:%M:%S
%s	Epoch seconds
%z / %Z	Offset / timezone name

Seq	Meaning
%V	ISO week
%a / %A	Weekday

Examples with Explanations

Now

```
date
date -u
date -Iseconds
date +%F
date +%Y%m%d-%H%M%S
date +%s
```

Convert / relative

```
date -d '2026-01-15 10:00'
date -d '2 days ago' +%F
date -d '@1700000000'
date -d 'next Monday'
```

Filenames and logs

```
logfile="app-$(date +%Y%m%d).log"
echo "$(date -Is) started" >> "$logfile"
```

File mtime

```
date -r /etc/hosts
stat -c %y /etc/hosts
```

Timezone one-off

```
TZ=UTC date
TZ=America/New_York date
timedatectl
```

Set clock (prefer NTP)

```
# last resort; prefer chrony/systemd-timesyncd
sudo date -s '2026-08-04 12:00:00'
sudo timedatectl set-time '2026-08-04 12:00:00'
sudo timedatectl set-ntp true
```

GNU vs scripts

```
# portable-ish epoch:
date +%s
# ISO local:
date -Iseconds
```

Notes / Pitfalls

- Setting time on production without NTP understanding breaks certs, logs, and Kerberos.
- Locale affects names (%B); use `LC_ALL=C` for stable parsing.
- macOS `date` flags differ; this book targets GNU/Linux.
- Leap seconds / monotonic clocks: for durations prefer `EPOCHREALTIME` or dedicated monotonic APIs in apps.
- Containers may use host time namespace.

2026-relevant notes

- Always enable NTP via `timedatectl` / `chrony` on servers.
- Log pipelines prefer UTC (`date -u -Iseconds`).
- For scheduling, pair with `systemd-run` / cron and explicit TZ.

Related Commands

- `timedatectl` — timezone and NTP status
- `hwclock` — hardware clock
- `cal` — calendar
- `touch -d` — set file times
- `stat` — file timestamps

Additional Resources

- `man date`, `man timedatectl`

timedatectl

Overview

`timedatectl` views and changes the system clock, timezone, RTC settings, and NTP synchronization on **systemd** systems. Prefer it over ad-hoc `date -s` for persistent timezone and NTP policy. On Ubuntu Server, default NTP is often **systemd-timesyncd**; some hosts use **chrony** instead — enable one cleanly, not both fighting.

Syntax

```
timedatectl [options] [command]
```

Common Commands / Options

Command / option	Description
<code>status</code>	Show time, TZ, NTP state (default)
<code>show</code>	Machine-readable properties
<code>list-timezones</code>	All zone names
<code>set-timezone Zone</code>	Set timezone (e.g. <code>UTC</code> , <code>America/New_York</code>)
<code>set-time 'YYYY-MM-DD HH:MM:SS'</code>	Set clock manually (disable NTP first if required)
<code>set-ntp true\ false</code>	Enable/disable NTP (systemd unit)
<code>set-local-rtc 0\ 1</code>	RTC in UTC (0, recommended for Linux) or local
<code>-H host / --machine=</code>	Remote/container via systemd
<code>--no-ask-password</code>	Non-interactive privilege

Key Use Cases

1. First-boot timezone + NTP setup
2. Verify clock sync before Kerberos/TLS-sensitive work
3. Switch TZ for multi-region servers (prefer UTC on servers)
4. Diagnose “TLS not yet valid” / log timestamp skew

Examples with Explanations

Status

```
timedatectl
timedatectl status
```

Shows local time, universal time, RTC, timezone, and whether NTP is active/synchronized.

Machine-readable

```
timedatectl show
timedatectl show -p Timezone -p NTP -p NTPSynchronized --value
```

Good for scripts and monitoring checks.

List and set timezone

```
timedatectl list-timezones | grep -i america
timedatectl list-timezones | grep -i '^Asia/'
sudo timedatectl set-timezone UTC
sudo timedatectl set-timezone America/New_York
timedatectl
```

Server fleets usually standardize on **UTC**; set human TZ only when local wall time is required.

Enable NTP (systemd-timesyncd path)

```
sudo timedatectl set-ntp true
timedatectl
systemctl status systemd-timesyncd --no-pager
# Ubuntu may use:
systemctl status systemd-timesyncd.service
```

`set-ntp true` lets systemd enable the appropriate time-sync unit.

Chrony hosts

```
# If chrony is the chosen stack:
sudo apt install chrony
sudo systemctl enable --now chrony
# timedatectl should still report NTP synchronized when chrony steps/slews correctly
chronyc tracking
```

```
chronyc sources -v
timedatectl
```

Don't leave timesyncd and chrony both active and conflicting — pick one source of truth.

Manual time (maintenance / air-gap)

```
sudo timedatectl set-ntp false
sudo timedatectl set-time '2026-08-09 12:00:00'
# re-enable NTP when network is available:
sudo timedatectl set-ntp true
```

Manual set is a last resort; large steps can confuse databases and cert validation.

RTC in UTC (recommended on Linux)

```
timedatectl | grep -i rtc
sudo timedatectl set-local-rtc 0
```

Dual-boot with Windows may fight over local vs UTC RTC — document the choice on shared hardware.

Verify sync health

```
timedatectl | grep -iE 'ntp|time zone|synchronized'
# timesyncd:
timedatectl timesync-status 2>/dev/null || true
journalctl -u systemd-timesyncd -n 20 --no-pager
# chrony:
chronyc tracking 2>/dev/null || true
```

“System clock synchronized: yes” is the headline for TLS and distributed systems.

date interaction

```
date
date -u
date --iso-8601=seconds
timedatectl
```

date formats the current clock; timedatectl manages policy (TZ/NTP/RTC).

Understanding Output

Typical `timedatectl` status fields:

Field	Meaning
Local time	Wall time in current TZ
Universal time	UTC
RTC time	Hardware clock reading
Time zone	IANA zone name + offset
System clock synchronized	Whether systemd believes sync is good
NTP service	active/inactive
RTC in local TZ	n = UTC (preferred on Linux-only)

`timedatectl show` exposes properties like `Timezone=`, `NTP=`, `NTPSynchronized=`, `TimeUsec=`.

Notes & Pitfalls

- Prefer NTP over permanent manual `set-time` on networked hosts.
- Containers often inherit host time and **cannot** change the host clock; TZ can still be set in-process via TZ env.
- Cloud images may already enforce UTC + timesync — cloud-init can reapply settings.
- Large backward clock jumps break `make`, systemd units, and DB leases — prefer slewing via NTP.
- Kerberos and TLS are intolerant of multi-minute skew.
- `set-timezone` does not rewrite historical log timestamps already written.
- Virtual hardware RTC quality varies; NTP remains mandatory for accuracy.

Related Commands

- `date` — print/format time
- `hwclock` — read/set hardware clock directly
- `chronyc` / `chronyd` — chrony stack
- `ntpdate` — legacy one-shot (avoid on modern systemd; use `chrony/timesyncd`)
- `hostnamectl` — host identity companion on first boot
- `timedatectl timesync-status` — timesyncd detail when available

Additional Resources

- `man timedatectl`
- `man systemd-timesyncd.service`
- `man chrony.conf` (when using chrony)

cal

Overview

`cal` displays a calendar in the terminal — current month by default, or a specific month/year. Handy for quick date context, scripting reminders, and verifying week boundaries. Some systems ship `ncal` with alternate layouts.

Syntax

```
cal [options] [[[day] month] year]
cal [options] [[month] year]
cal [options] -y [year]
```

Exact syntax variants exist between util-linux `cal` and BSD-style `cal/ncal`.

Common Options (util-linux / common)

Option	Description
<code>-1 / -3</code>	One month / three months
<code>-y</code>	Whole year
<code>-m</code>	Monday as first day of week
<code>-s</code>	Sunday first (when supported)
<code>-w</code>	Week numbers
<code>-j</code>	Julian day numbers
<code>-A n / -B n</code>	Months after / before
<code>-h</code>	Highlight off (or help, depending on version)
<code>--color</code>	Colorize today (newer)

Check `cal --help` on your host.

Examples with Explanations

Current month

```
cal
cal -3
cal -m
cal -w
```

Specific month / year

```
cal 12 2026
cal 2026
cal -y 2026
```

Week numbers for planning

```
cal -w
cal -mw 9 2026
```

Surrounding months

```
cal -A 1 -B 1
cal -3
```

Historical curiosity

```
cal 9 1752                                     # UK calendar reform month (implementation-dependent)
```

Scripts: parse carefully

```
# better for machine dates:
date +%F
date +%V                                     # ISO week
# use cal for human display only
```

ncal alternative

```
ncal
ncal -b
ncal -w
```

Notes / Pitfalls

- Highlighting “today” depends on terminal and build options.
- Week-start Monday/Sunday differs by locale and flags — be explicit with `-m` when it matters.
- Do not parse `cal` output for business logic; use `date` and proper libraries.
- Year range limits differ; very ancient/future years may error.
- BusyBox `cal` is minimal.

2026-relevant notes

- ISO week numbers (`date +%V`, `cal -w`) still matter for release trains and on-call rotations.
- Prefer `date` for automation; keep `cal` as a human aid over SSH.
- Terminal Unicode width issues can misalign rare locales.

Related Commands

- `date` — current date/time formatting
- `ncal` — alternate calendar view
- `timedatectl` — timezone/NTP status
- `calendar` — reminder file tool (if installed)
- `hwclock` — hardware clock

Additional Resources

- `man cal`, `man ncal`

whoami

Overview

`whoami` prints the **effective** username of the current user. It is a tiny identity check for scripts and for noticing when `sudo/su` changed who you are. For full identity (uid, gid, groups) use `id`. For login name from the controlling terminal, see `logname` (different edge cases).

Syntax

```
whoami [OPTION]
# usually no options needed
```

Common Options

Option	Description
<code>--help</code>	Help
<code>--version</code>	Version

Key Use Cases

1. Confirm effective user in a shell
2. Guard scripts that must (or must not) run as root
3. Annotate logs with actor name
4. Verify `sudo -u` targets

Examples with Explanations

Basics

```
whoami
sudo whoami          # usually root
sudo -u nobody whoami
su - alice -c whoami
```

Script guards

```
if [ "$(whoami)" != root ]; then
    echo "run as root" >&2
    exit 1
fi

# prefer numeric for reliability:
if [ "$(id -u)" -ne 0 ]; then
    echo "run as root" >&2
    exit 1
fi
```

Contrast with id / logname

```
whoami
id -un          # equivalent effective username
id
logname         # user logged on the tty (may differ)
```

After `sudo -s`, `whoami` is `root` while `logname` may still show the original login user.

Logging

```
echo "$(date -Is) user=$(whoami) action=deploy" >> deploy.log
```

Containers

```
whoami
id
# USER instruction in Dockerfile determines default
```

Notes / Pitfalls

- Reflects **effective** ids (setuid binaries can change this).
- Prefer `id -u` for root checks — numeric, locale-proof.
- Not a substitute for authentication/authorization in applications.
- May print **I have no name!** style issues only via related tools when `passwd` entry missing; `whoami` uses libc name lookup.

2026-relevant notes

- In Kubernetes/pods, check whether the container runs as non-root (`whoami` / `id`).
- Rootless Podman maps users — names inside may differ from host.
- Automation should key off uid numbers stored in configs, not display names alone.

Related Commands

- `id` — full identity
- `logname` — login name from utmp
- `who` / `w` — logged-in sessions
- `sudo` / `su` — change user
- `getent passwd` — account database

Additional Resources

- `man whoami`

id

Overview

`id` prints the real and effective user and group IDs for a process (by default, yours), plus supplementary groups. It is the authoritative quick identity tool for permissions debugging — better than `whoami` alone when groups and numeric IDs matter.

Syntax

```
id [options] [user]
```

Common Options

Option	Description
<code>-u, --user</code>	Print only user ID
<code>-g, --group</code>	Print only primary group ID
<code>-G, --groups</code>	Print all group IDs
<code>-n, --name</code>	Print names instead of numbers (with <code>-u/-g/-G</code>)
<code>-r, --real</code>	Real ID instead of effective
<code>-z, --zero</code>	NUL-terminate (with <code>-G</code> etc.)
<code>-A</code>	Security context extras on some systems
<code>-Z</code>	SELinux context (when available)

Examples with Explanations

Default

```
id
# uid=1000(alice) gid=1000(alice) groups=1000(alice),27(sudo),100(users)
```

Specific fields

```
id -u
id -un
id -g
id -gn
id -G
id -Gn
```

Other users

```
id www-data
id -un root
getent passwd www-data
```

Root check in scripts

```
if [ "$(id -u)" -ne 0 ]; then
    echo "need root" >&2
    exit 1
fi
```

Real vs effective (setuid context)

```
id -u
id -ru
id -u -r
# if they differ, effective privileges differ from real user
```

SELinux

```
id -Z
```

Format for logs

```
printf 'User: %s (UID: %d)\n' "$(id -un)" "$(id -u)"
echo "groups=$(id -Gn)"
```

Permission debugging pattern

```
id
namei -l /path/to/resource
ls -l /path/to/resource
getfacl /path/to/resource 2>/dev/null
```

Notes / Pitfalls

- Names come from NSS (`passwd/group/sss/ldap`) — offline LDAP can make lookups hang.
- Prefer numeric IDs in scripts for comparisons; names can rename.
- Supplementary groups affect file access; primary gid alone is incomplete.
- Containers may show only a subset of host groups or remapped UIDs.
- `id user` looks up that account; default is the calling process credentials.

2026-relevant notes

- Systemd user sessions and linger affect available groups for some services — debug with `id` inside the unit’s environment (`systemd-run --user`).
- Rootless containers: high UIDs on the host map to low UIDs inside.
- For policy (“must be in group docker”), check `id -nG | tr ' ' '\n' | grep -x docker`.

Related Commands

- `whoami` — effective username only
- `groups` — group names shorthand
- `getent` — account database lookup
- `newgrp` / `sg` — run with alternate group
- `sudo` / `su` — change identity
- `getfacl` — ACL permissions

Additional Resources

- `man id`

who

Overview

who shows who is logged in, with terminal, login time, and sometimes host/remote info from utmp/wtmp accounting. Related tools: **w** (richer: load + what they run), **users** (names only), **last** (historical logins).

Syntax

```
who [options] [file]
who am i
whoami                # different command: effective user
```

Common Options

Option	Description
-a	All information (combo flags)
-b	Boot time
-d	Dead processes
-H	Header
-l	Login processes
-m	Same as who am i (hostname and user for stdin)
-q	Count and names only
-r	Runlevel (legacy)
-s	Short format
-t	Last system clock change
-u	Idle time / PID
-T	Message status of mesg (+/-)

Optional **file** defaults to utmp (often **/var/run/utmp** or **/run/utmp**).

Examples with Explanations

Who is on the system

```
who
who -H
who -u
who -q
```

Boot time

```
who -b
uptime -s
```

Yourself

```
who am i
who -m
logname
```

Compare with w

```
who
w
users
```

w adds load averages and the current command line of each session.

Historical

```
last
last -a
who /var/log/wtmp # if supported / accessible
```

Scripting inventory

```
who | awk '{print $1}' | sort -u
```

Notes / Pitfalls

- utmp accuracy varies: display managers, wayland, lingering sessions, and containers may not appear as classic ttys.
- SSH multiplexing and tmux can make “one human, many lines”.
- Empty output on some minimal containers is normal — no utmp traffic.
- Privacy: listing logins is normal for admins; still treat usernames carefully in logs.
- `whoami` `who am i`.

2026-relevant notes

- Prefer `loginctl` on `systemd` for modern session management views.
- Audit pipelines use `last`/`journalctl` more than live `who`.
- Remote access gateways may hide real client IPs behind proxies — check `sshd` logs.

Related Commands

- `w` — richer logged-in view
- `users` — names only
- `last` / `lastlog` — history
- `logname` — login name
- `loginctl` — `systemd` sessions
- `whoami` / `id` — identity of current process

Additional Resources

- `man who`

W

Overview

w shows who is logged in **and** what they are doing, plus system uptime and load averages. It combines aspects of **uptime**, **who**, and a glimpse of each session's current process — a classic first glance after SSH'ing into a busy server.

Syntax

```
w [options] [user]
```

Common Options

Option	Description
-h	No header
-s	Short format (less detail)
-f	Toggle from field (remote host)
-i	Show IP instead of hostname when possible
-o	Old-style output order
-u	Ignore username for current process / idle calc variants
user	Show only this user

Examples with Explanations

Default

```
w
# 10:22:01 up 5 days,  3:01,  4 users,  load average: 0.42, 0.35, 0.30
# USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
# alice     pts/0    203.0.113.10    09:01    0.00s  0.12s  0.01s w
```

Filter one user

```
w alice
```

Short / no header

```
w -h  
w -s
```

IP addresses

```
w -i
```

Correlate with load tools

```
w  
uptime  
ps -u alice  
top -bn1 | head
```

When you need history

```
w  
last -a | head  
journalctl _COMM=sshd -n 20
```

Understanding columns

Column	Meaning
USER	Login name
TTY	Terminal device
FROM	Remote host (if any)
LOGIN@	Login time
IDLE	Idle time on tty
JCPU	Time of all processes on tty
PCPU	Time of current process
WHAT	Current command line

Exact columns vary slightly by implementation (procps-ng).

Notes / Pitfalls

- **WHAT** is a snapshot — can be misleading for short-lived commands.
- Idle time does not mean CPU idle for the whole system.
- Containers and GUI sessions may not appear as classic pts entries.
- Do not rely on `w` alone for security auditing; use auth logs/journal.
- High load with few users often means services/cron, not interactive humans.

2026-relevant notes

- Still excellent for quick SSH triage; pair with `systemd-cgtop` / `bttop` for service-heavy hosts.
- `loginctl list-sessions` gives systemd's session view.
- Cloud bastions: many **FROM** IPs may be the bastion, not the end user.

Related Commands

- `who` — simpler login list
- `uptime` — load + uptime only
- `ps` / `top` / `htop` — process detail
- `last` — login history
- `loginctl` — systemd sessions
- `finger` — legacy user info (if installed)

Additional Resources

- `man w`

last

Overview

`last` shows a history of logins, reboots, and runlevel changes from the `wtmp` accounting file (typically `/var/log/wtmp`). Use it after security incidents, to see when a machine rebooted, or who logged in from where. Companion: `lastb` (bad logins from `btmp`), `lastlog` (per-user last login).

Syntax

```
last [options] [username...] [tty...]
```

Common Options

Option	Description
<code>-a</code>	Display hostname in last column
<code>-d</code>	Translate IP to hostname
<code>-F</code>	Full timestamps
<code>-n N, -N</code>	Show N lines
<code>-R</code>	Suppress hostname field
<code>-i</code>	Show IP as numbers
<code>-x</code>	Include shutdown/runlevel entries
<code>-w</code>	Widen fields for long names
<code>-f file</code>	Alternate wtmp file
<code>-s / -t</code>	Since / until times (when supported)

Examples with Explanations

Recent logins

```
last
last -n 20
last -a
last -i
```

```
last -F
```

One user

```
last alice  
last root
```

Reboots and shutdowns

```
last reboot  
last -x | head  
who -b  
uptime -s
```

Bad logins

```
sudo lastb | head  
sudo lastb -a
```

Requires permission to read `btm`.

Rotate awareness

```
last -f /var/log/wtmp.1  
ls /var/log/wtmp*
```

After logrotate, older history lives in rotated files.

SSH audit starter

```
last -ai | head -50  
journalctl -u ssh -n 50  
journalctl _COMM=sshd -n 50
```

Notes / Pitfalls

- Missing/empty wtmp means no history (containers, stripped images, misconfigured logging).
- Hostnames from `-d` depend on DNS and can mislead; prefer `-i` for forensics.
- Entries can be incomplete if the machine crashed without clean logout (shows `crash / down` / still logged in).
- Not a full security log — pair with `journalctl`, `auditd`, cloud provider logs.
- Timezones: interpret with system TZ settings.

2026-relevant notes

- Many fleets centralize auth logs; local `last` is still valuable on a single host incident.
- `systemd` / `logind` sessions: also check `loginctl` and `journal`.
- Immutable hosts may reset wtmp on reboot depending on design — know your image.

Related Commands

- `lastb` — failed logins
- `lastlog` — per-user last login table
- `who` / `w` — currently logged in
- `journalctl` — structured logs
- `utmpdump` — raw utmp/wtmp decode
- `aureport` — audit framework reports (if `auditd`)

Additional Resources

- `man last`, `man wtmp`

env

Overview

`env` runs a program in a modified environment, or prints the current environment variables when given no command. It is essential for shebangs (`#!/usr/bin/env bash`), clearing the environment (`-i`), and injecting variables for a single command without exporting them permanently in the shell.

Syntax

```
env [options] [NAME=VALUE]... [command [args...]]
env                                     # print environment
```

Common Options

Option	Description
<code>-i, --ignore-environment</code>	Start with an empty environment
<code>-u NAME, --unset=NAME</code>	Remove variable
<code>-0, --null</code>	NUL-delimit output
<code>-C dir, --chdir=dir</code>	Change directory before exec (newer coreutils)
<code>-S string</code>	Split string on shebang lines (see man)
<code>-v</code>	Verbose
<code>--</code>	End of options

Examples with Explanations

Print environment

```
env
env | sort
env | grep -E '^(PATH|HOME|USER)='
printenv PATH          # single var alternative
```

Run with extra variables

```
env FOO=bar BAZ=1 ./script.sh
env NODE_ENV=production node app.js
```

Equivalent to shell `FOO=bar ./script.sh` for simple cases; `env` shines with cleared environments and shebangs.

Clear environment

```
env -i PATH=/usr/bin:/bin HOME="$HOME" bash --noprofile --norc
env -i $(cat allowed.env | xargs) command
```

Useful for clean builds and debugging “works in my shell” issues.

Unset for one command

```
env -u http_proxy -u https_proxy curl -I https://example.com
```

Shebang portability

```
#!/usr/bin/env bash
#!/usr/bin/env python3
```

`env` locates the interpreter via `PATH`, avoiding hard-coded `/usr/bin/python3` when layouts differ. (Security-sensitive contexts may prefer absolute paths.)

Chdir then run (when supported)

```
env -C /var/www -i PATH=/usr/bin:/bin ./deploy.sh
```

NUL-safe dump

```
env -0 | tr '\0' '\n' | sort
```


Notes / Pitfalls

- `env -i` is easy to over-strip — many programs need `PATH`, `HOME`, `TERM`, `LANG`.
- Shebang line length limits exist; complex `env -S` usage has edge cases.
- Printing `env` may leak secrets (`AWS_*`, tokens) — avoid pasting full dumps.
- Order of assignment and `-u` matters; see man page for processing order.
- Shell builtins are not found by `env` — it execs real programs from `PATH`.

2026-relevant notes

- Container entrypoints often use `env` or `exec` forms to inject config.
- Prefer explicit `.env` loading tools or `systemd Environment=` for services.
- Security: `LD_PRELOAD` and similar can be injected via `env` — hardened systems scrub them for `setuid`.

Related Commands

- `printenv` — print variables
- `export` / `set` — shell environment
- `envdir` / `dotenv` — external `env` loaders
- `systemd-run` / `unit Environment=` — service `env`
- `sudo -E` / `sudo --preserve-env` — elevated `env` policy

Additional Resources

- `man env`

free

Overview

`free` summarizes system memory usage: physical RAM and swap. Output comes from `/proc/meminfo`. Modern `free` includes an **available** estimate (memory reclaimable without swapping) that is more useful than the old “free only” column for deciding if a host is under memory pressure.

Syntax

```
free [options]
```

Common Options

Option	Description
<code>-h, --human</code>	Human-readable units
<code>-b / -k / -m / -g</code>	Bytes / KiB / MiB / GiB
<code>-t</code>	Total line
<code>-s N</code>	Refresh every N seconds
<code>-c N</code>	Repeat N times (with <code>-s</code>)
<code>-w</code>	Wide output (separate buffers/cache on older styles)
<code>-v</code>	Version
<code>--si</code>	Powers of 1000 instead of 1024

Examples with Explanations

Everyday

```
free -h
free -m
free -h -t
```

Watch live

```
free -h -s 2
watch -n1 free -h
```

Scripts

```
free -b | awk '/^Mem:/ {print $2, $7}' # total, available (column positions can shift - ve
# safer parse of meminfo:
awk '/MemAvailable:/ {print $2}' /proc/meminfo
```

Interpret with pressure tools

```
free -h
cat /proc/pressure/memory 2>/dev/null # PSI on modern kernels
vmstat 1 5
swapon --show
```

Containers / cgroups

```
free -h
# may show host memory unless constrained views; check cgroup limits:
cat /sys/fs/cgroup/memory.max 2>/dev/null
cat /sys/fs/cgroup/memory.current 2>/dev/null
```

Understanding the columns

Typical modern output:

	total	used	free	shared	buff/cache	available
Mem:	...					
Swap:	...					

Field	Meaning
total	Installed / visible memory
used	In use (implementation-defined accounting)
free	Completely unused
shared	tmpfs / shared memory estimate
buff/cache	Kernel buffers and page cache
available	Estimate of memory available for new workloads

Field	Meaning
-------	---------

Linux uses free RAM for cache — low “free” with high “available” is often healthy.

Notes / Pitfalls

- Do not panic solely because **free** is low; watch **available**, reclaim, PSI, and OOM logs.
- Swap usage can be normal for infrequently touched pages; heavy swap I/O is the problem.
- Column layout changed over the years — verify before parsing.
- Huge pages, zswap, and compressed RAM affect interpretation.
- VMs: balloon drivers and host overcommit complicate the picture.

2026-relevant notes

- Prefer **PSI** (`/proc/pressure`) and metrics backends for production memory SLOs.
- `systemd-cgtop` and `cgroup v2` memory stats matter more than host **free** inside dense multi-service hosts.
- OOM: `journalctl -k | grep -i oom / dmesg` for killer events.

Related Commands

- `vmstat` — memory + runqueue + io
- `top` / `htop` / `btm` — per-process memory
- `swapon` / `swapoff` — swap devices
- `cat /proc/meminfo` — raw counters
- `smem` — proportional set size (if installed)
- `slabtop` — kernel slab usage

Additional Resources

- `man free`, `man proc`

lscpu

Overview

lscpu displays CPU architecture information from sysfs and `/proc/cpuinfo`: sockets, cores, threads, model name, caches, virtualization flags, NUMA layout, and vulnerabilities mitigations summary. Use it before tuning parallelism, interpreting load averages, or choosing binary architectures.

Syntax

```
lscpu [options]
```

Common Options

Option	Description
<code>-a / -b / -c</code>	Online/offline CPU filtering variants
<code>-e</code>	Extended parsable table of CPUs
<code>-p</code>	Parsable (key,value or CSV-like)
<code>-J</code>	JSON output (newer util-linux)
<code>-x</code>	Hex masks
<code>-y</code>	Physical instead of logical in some fields
<code>-s</code>	Sysroot for inspection
<code>-B</code>	Byte sizes raw

Examples with Explanations

Overview

```
lscpu
```

Parseable / JSON

```
lscpu -p
lscpu -J | jq .
lscpu -e=cpu,node,socket,core,online
```

Core counts for scripts

```
nproc
lscpu | awk -F: '/^CPU\s\):/ {gsub(/ /,"",$2); print $2}'
# prefer:
nproc --all
getconf _NPROCESSORS_ONLN
```

Virtualization and flags

```
lscpu | grep -E 'Virtualization|Hypervisor|Flags|Model name|Thread|Core|Socket'
grep -m1 flags /proc/cpuinfo
```

Look for vmx (Intel) / svm (AMD) for hardware virtualization; ht for hyperthreading.

NUMA

```
lscpu | grep -i numa
numactl -H 2>/dev/null
ls /sys/devices/system/node
```

Security mitigations summary

```
lscpu | sed -n '/Vulnerabilities:/$p'
# detailed:
grep . /sys/devices/system/cpu/vulnerabilities/*
```

Containers

```
lscpu
nproc
# CPU affinity may be constrained by cgroup
cat /sys/fs/cgroup/cpuset.cpus.effective 2>/dev/null
```

Notes / Pitfalls

- CPU(s) counts logical CPUs (hardware threads), not physical cores.
- Hypervisors can present fake topology; verify when licensing by socket/core.
- Offline CPUs and hotplug change counts — re-run after changes.
- Flags strings are long; pipe to **grep** for specific features (**avx2**, **aes**, ...).
- Don't use load average without knowing logical CPU count.

2026-relevant notes

- ARM (aarch64) and RISC-V hosts are common; scripts must not assume x86_64.
- JSON output helps inventory agents.
- Mitigations still affect performance; compare with workload benchmarks, not only flag presence.

Related Commands

- **nproc** — online processor count
- **lshw -class processor** — alternate inventory
- **cat /proc/cpuinfo** — per-cpu raw
- **numactl** — NUMA policy
- **taskset** — CPU affinity
- **tuned / power profiles** — CPU governors

Additional Resources

- **man lscpu**

lsmem

Overview

The `lsmem` command displays information about memory ranges and their online/offline status. It's particularly useful for systems with memory hotplug capabilities and NUMA architectures.

Syntax

```
lsmem [options]
```

Common Options

Option	Description
<code>-a</code>	List all memory ranges
<code>-b</code>	Show output in bytes
<code>-h</code>	Human-readable sizes
<code>-o columns</code>	Specify output columns
<code>-r</code>	Raw output format
<code>-S</code>	Split by node
<code>-s</code>	Show summary only
<code>--sysroot=dir</code>	Use alternative sysfs root

Output Columns

Column	Description
RANGE	Memory address range
SIZE	Size of memory range
STATE	Online/Offline status
REMOVABLE	Whether memory can be removed
BLOCK	Memory block number
NODE	NUMA node number
ZONES	Memory zones

Key Use Cases

1. Memory inventory
2. NUMA topology analysis
3. Memory hotplug management
4. System capacity planning
5. Memory troubleshooting

Examples with Explanations

Example 1: Basic Memory Information

```
lsmem
```

Shows memory ranges and their status

Example 2: Human-Readable Sizes

```
lsmem -h
```

Displays memory sizes in human-readable format (KB, MB, GB)

Example 3: Summary Only

```
lsmem -s
```

Shows only summary information

Understanding Memory States

State	Description
online	Memory is available for use
offline	Memory is not available
going-offline	Memory is being taken offline
going-online	Memory is being brought online

Memory Block Management

Memory is managed in blocks: - Block size typically 128MB or 256MB - Blocks can be individually onlined/offlined - Useful for memory hotplug operations

Common Usage Patterns

1. Check total memory:

```
lsmem -s | grep "Total online memory"
```

2. List offline memory:

```
lsmem | grep offline
```

3. Show NUMA distribution:

```
lsmem -S
```

NUMA Memory Information

For NUMA systems: - Shows memory distribution across nodes - Helps with memory locality optimization - Useful for performance tuning

Memory Hotplug Operations

1. Check removable memory:

```
lsmem | grep "yes" | grep "REMOVABLE"
```

2. Find offline blocks:

```
lsmem -a | awk '$3=="offline" {print $4}'
```

Performance Analysis

Memory information useful for: - Memory bandwidth optimization - NUMA-aware application tuning - Memory pressure analysis - Capacity planning

Related Commands

- `free` - Memory usage statistics
- `cat /proc/meminfo` - Detailed memory information
- `numactl --hardware` - NUMA topology
- `dmidecode -t memory` - Physical memory information
- `lshw -C memory` - Hardware memory details

Additional Resources

- [lsmem Manual](#)
- [Memory Management Guide](#)

Best Practices

1. Monitor memory hotplug status
2. Understand NUMA topology
3. Check removable memory before maintenance
4. Use with other memory analysis tools
5. Consider memory block alignment

Scripting Examples

1. Count online memory blocks:

```
lsmem | grep -c "online"
```

2. Get total memory size:

```
lsmem -s | grep "Total online memory" | awk '{print $4}'
```

3. Check NUMA nodes:

```
lsmem -o NODE | grep -v NODE | sort -u
```

Memory Zones

Common memory zones: - **DMA**: Direct Memory Access zone - **DMA32**: 32-bit DMA zone - **Normal**: Regular memory zone - **HighMem**: High memory zone (32-bit systems) - **Movable**: Memory that can be migrated

System Integration

1. Memory monitoring:

```
watch -n 5 'lsmem -s'
```

2. NUMA optimization:

```
lsmem -S | grep "node 0"
```

3. Capacity reporting:

```
echo "Total Memory: $(lsmem -s | grep 'Total online' | awk '{print $4}')
```

Troubleshooting

1. Memory not showing up
2. Offline memory blocks
3. NUMA node misalignment
4. Memory hotplug failures
5. Inconsistent memory reporting

Advanced Usage

1. Custom column output:

```
lsmem -o RANGE,SIZE,STATE,NODE
```

2. Raw format for parsing:

```
lsmem -r
```

3. Alternative sysfs root:

```
lsmem --sysroot=/alternative/path
```

Memory Block Operations

To manage memory blocks (requires root):

```
# Online a memory block
echo online > /sys/devices/system/memory/memory64/state

# Offline a memory block
echo offline > /sys/devices/system/memory/memory64/state
```

Integration Examples

1. With NUMA tools:

```
lsmem -S && numactl --hardware
```

2. Memory pressure monitoring:

```
lsmem -s && free -h
```

3. System inventory:

```
echo "Memory Layout:" && lsmem -h
```

Output Formatting

1. Specific columns:

```
lsmem -o SIZE,STATE | column -t
```

2. Summary with details:

```
lsmem -s && echo "---" && lsmem
```

3. Node-specific information:

```
lsmem | awk '$6==0 {print}' # Node 0 only
```

lsusb

Overview

`lsusb` lists USB devices connected to the system (buses, IDs, vendor/product strings). Essential for debugging docks, keys, serial adapters, and whether a device is even enumerated before loading drivers.

Package: often `usbutils`.

Syntax

```
lsusb [options]
```

Common Options

Option	Description
<code>-v, --verbose</code>	Detailed descriptors
<code>-vv</code>	Even more verbose
<code>-t</code>	Tree view
<code>-s [[bus]:][devnum]</code>	Show only specific device
<code>-d vendor:product</code>	Filter by ID
<code>-D device</code>	Query device node under <code>/dev/bus/usb</code>
<code>-k</code>	Kernel driver binding info (when supported)

Examples with Explanations

List devices

```
lsusb
lsusb -t
```

Filter

```
lsusb -d 0781:5567
lsusb -s 001:005
```

Verbose probe

```
sudo lsusb -v | less
sudo lsusb -v -d 1d6b:0002 | less
```

Correlate with kernel messages

```
lsusb
dmesg -w
# plug device, watch enumeration
journalctl -k -f
```

Find device nodes

```
ls -l /dev/bus/usb/*/*
ls -l /dev/ttyUSB* /dev/ttyACM* 2>/dev/null
```

Tree + drivers

```
lsusb -t
# shows hubs and nesting; helpful for power/hub issues
```

Notes / Pitfalls

- Permission: some verbose details need root.
- Unauthorized USB policy (USBGuard) may block devices that still appear partially.
- VMs need USB passthrough configured; host `lsusb` guest.
- IDs are hex vendor:product from the USB-IF database; strings can be generic.
- Unstable enumeration often means cable/power/hub issues, not software.

2026-relevant notes

- USB-C docks create deep hub trees — use `-t`.
- `usbguard`, `udev` rules, and secure boot policies affect usability more than `lsusb` itself.
- For serial adapters, also check `dmesg` for `ttyUSB/ttyACM` assignment.

Related Commands

- `lspci` — PCI devices
- `lshw` — hardware inventory
- `dmesg` / `journalctl -k` — kernel enumeration logs
- `udevadm info` — device metadata
- `usb-devices` — alternate text dump
- `modprobe` — load drivers

Additional Resources

- `man lsusb`

lsmod

Overview

lsmod lists loaded **kernel modules** (name, size, use count, dependents). It is a thin, readable view of `/proc/modules`. Use it when debugging drivers, missing hardware support, or before `modprobe/rmmod` operations.

Syntax

```
lsmod
# no significant options - see modprobe/rmmod for management
```

Examples with Explanations

List modules

```
lsmod
lsmod | sort
lsmod | head
```

Search for a driver

```
lsmod | grep -i usb
lsmod | grep -i nvidia
lsmod | grep -E '^(nfs|overlay|br_netfilter)'
```

Dependency / use count

```
lsmod | awk 'NR==1 || $3>0 {print}'
# third column: used by count; Used by column shows dependents
```

Correlate with hardware

```
lsusb
lspci -k
lsmod | grep -i xhci
```

`lspci -k` shows which kernel driver is bound to a PCI device.

Module details

```
modinfo e1000e
modprobe -n -v e1000e      # dry-run
cat /proc/modules | grep e1000e
```

Unload safety check

```
lsmod | grep module_name
sudo modprobe -r module_name  # preferred over raw rmmod
# fails if in use - check dependents in lsmod
```

Blacklist context

```
lsmod | grep nouveau
# if you blacklisted a module, it should not appear after reboot
grep -r nouveau /etc/modprobe.d/ 2>/dev/null
```

Understanding columns

Column	Meaning
Module	Module name
Size	Memory size in bytes
Used by	Reference count and dependent module names

A non-zero use count means unload will fail until dependents release it.

Notes / Pitfalls

- Built-in kernel features are **not** modules — absence from `lsmod` doesn't mean “unsupported”.
- Out-of-tree modules (ZFS, some NVIDIA) still appear when loaded.
- Containers usually cannot load modules; operations need host privileges.
- Secure Boot may restrict unsigned modules.
- Don't randomly `rmmod` storage/network drivers on remote hosts.

2026-relevant notes

- Initramfs and UKI images embed critical modules — `lsmod` after boot is not the whole story.
- Prefer `modprobe` over `insmod` for dependency resolution.
- For policy, use `/etc/modules-load.d/` and `/etc/modprobe.d/` rather than ad-hoc loads.

Related Commands

- `modprobe` / `rmmod` / `insmod` — load/unload
- `modinfo` — module metadata
- `lspci -k` / `lsusb` — hardware binding
- `dmesg` — load errors
- `depmod` — rebuild module deps

Additional Resources

- `man lsmod`, `man 5 modules`

hwinfo

Overview

The `hwinfo` command provides comprehensive hardware information. It's a powerful tool that probes for hardware and displays detailed information about various hardware components.

Syntax

```
hwinfo [options]
```

Common Options

Option	Description
<code>--short</code>	Brief hardware list
<code>--arch</code>	Show architecture info
<code>--bios</code>	Show BIOS info
<code>--block</code>	Show block devices
<code>--cpu</code>	Show CPU info
<code>--disk</code>	Show disk devices
<code>--memory</code>	Show memory info
<code>--network</code>	Show network devices
<code>--pci</code>	Show PCI devices
<code>--usb</code>	Show USB devices
<code>--all</code>	Show all hardware
<code>--save-config</code>	Save hardware config
<code>--log file</code>	Write log to file

Key Use Cases

1. Hardware inventory
2. System diagnostics
3. Driver verification
4. System documentation
5. Troubleshooting

Examples with Explanations

Example 1: Brief Summary

```
hwinfo --short
```

Show brief hardware list

Example 2: CPU Info

```
hwinfo --cpu
```

Show detailed CPU information

Example 3: Storage Info

```
hwinfo --disk --short
```

Show brief disk information

Understanding Output

Categories of information: - System overview - CPU details - Memory configuration - Storage devices - Network interfaces - Peripheral devices

Common Usage Patterns

1. Full system scan:

```
hwinfo --all
```

2. Network check:

```
hwinfo --network
```

3. Save hardware info:

```
hwinfo --all --log hardware.log
```

Performance Analysis

- Comprehensive scanning
- Resource intensive
- Database lookups
- Device probing time
- Log file generation

Related Commands

- `lshw` - List hardware
- `lspci` - List PCI devices
- `lsusb` - List USB devices
- `dmidecode` - DMI table info
- `inxi` - System information

Additional Resources

- [Hardware Info Documentation](#)
- [Linux Hardware Guide](#)
- [System Information Tools](#)

Hardware Categories

1. Processors
2. Memory modules
3. Storage devices
4. Network adapters
5. Peripheral devices

Best Practices

1. Regular hardware audits
2. Document configurations
3. Monitor changes
4. Keep logs
5. Update hardware database

Processes and Jobs

Intro

Intro

List, prioritize, signal, and background work. Covers classic job control in the shell as well as system-wide process tools (`ps`, `top` family, `pgrep/pkill`).

Commands in this part

Command	Role
<code>ps</code>	<code>ps</code> snapshots processes.
<code>pstree</code>	<code>pstree</code> shows running processes as a tree, making parent/child relationships obvious — ideal for seeing what a...
<code>top</code>	<code>top</code> is the classic interactive process monitor: CPU, memory, and a live-sorted process table.
<code>htop</code>	<code>htop</code> is an interactive process viewer with a clearer UI than classic <code>top</code> : mouse support, tree view, easy...
<code>procs</code>	<code>procs</code> is a modern replacement for <code>ps</code> written in Rust, with colorized output, tree views, keyword highlighting, and...
<code>pgrep</code>	<code>pgrep</code> lists process IDs matching selection criteria (name, full command line, user, terminal, etc.).
<code>pidof</code>	<code>pidof</code> prints the PIDs of running programs matched by name.
<code>kill</code>	<code>kill</code> sends a signal to one or more processes by PID.
<code>killall</code>	<code>killall</code> sends a signal to all processes matching a command name.
<code>jobs</code>	<code>jobs</code> lists active jobs of the current shell: running background tasks, stopped jobs, and their job IDs.
<code>fg</code>	<code>fg</code> resumes a job in the foreground, giving it the terminal again (stdin/stdout/signals).

Command	Role
<code>bg</code>	<code>bg</code> resumes a stopped job in the background so it continues running without occupying the terminal.
<code>nohup</code>	<code>nohup</code> runs a command immune to <code>SIGHUP</code> , so it keeps running after you log out of a terminal session.
<code>timeout</code>	<code>timeout</code> runs a command and sends a signal if it still runs after a duration.
<code>watch</code>	<code>watch</code> runs a command repeatedly, showing fullscreen output so you can see changes over time.
<code>sleep</code>	<code>sleep</code> suspends execution for a specified time.
<code>nice</code>	<code>nice</code> starts a command with a modified CPU niceness — a soft priority bias for the scheduler.
<code>renice</code>	The <code>renice</code> command alters the scheduling priority of running processes.
<code>ionice</code>	<code>ionice</code> sets or queries the I/O scheduling class and priority of a process (Linux CFQ/BFQ-oriented interface;...
<code>fuser</code>	<code>fuser</code> lists PIDs using a file, mount point, or network port.
<code>ulimit</code>	<code>ulimit</code> is a shell builtin (<code>bash/zsh/etc.</code>) that shows or sets resource limits for the current shell and its child...
<code>ipcs</code>	<code>ipcs</code> lists System V IPC objects: message queues, shared memory segments, and semaphore arrays.
<code>ipcrm</code>	<code>ipcrm</code> removes System V IPC objects: shared memory segments, semaphore arrays, and message queues.

Suggested starting points

1. Inventory: `ps`, `pstree`, then interactive `top`/`htop`/`bttop`.
2. Select by name: `pgrep` / `pidof` before `kill`/`killall`.
3. Job control in one shell: `jobs`, `fg`, `bg`, `nohup`.
4. Priority and I/O class: `nice`, `renice`, `ionice`.
5. Limits and IPC: `ulimit`, `ipcs`/`ipcrm`, `fuser`.

Related parts

- System monitoring — deeper CPU/I/O tracing
- Services and runtime — prefer `systemctl` for daemons
- Shell commands — `time` while iterating

Continue with the individual command pages in this part.

ps

Overview

`ps` snapshots processes. Linux `ps` accepts both **BSD-style** (`ps aux`) and **UNIX-style** (`ps -ef`) option sets — mixing styles works but confuses readers; pick one family per invocation. For a live view use `top`/`htop`; for PID selection by name prefer `pgrep`.

Syntax

```
ps [options]
```

Common Invocations

Form	Description
<code>ps aux</code>	BSD: all users, CPU/MEM columns
<code>ps -ef</code>	UNIX: full listing
<code>ps -u USER</code>	One user's processes
<code>ps -p PID</code>	Specific PID(s)
<code>ps -C name</code>	By command name
<code>ps --forest / f</code>	Tree view
<code>ps -eo ...</code>	Custom columns

Useful selectors: `-o pid,ppid,user,pcpu,pmem,stat,etime,cmd, --sort=-%mem`.

Key Use Cases

1. Identify heavy CPU/memory consumers
2. Find PIDs before signaling
3. Inspect command lines and parentage
4. Script process inventory / monitoring checks

Examples with Explanations

Classic full lists

```
ps aux | head
ps -ef | head
```

`aux` is the muscle-memory form on Linux; `-ef` is common in SysV docs.

Top memory and CPU

```
ps aux --sort=-%mem | head -n 15
ps aux --sort=-%cpu | head -n 15
```

Snapshot only — a process can spike between samples.

Custom columns

```
ps -eo pid,ppid,user,%cpu,%mem,stat,etime,cmd --sort=-%cpu | head
```

`etime` is elapsed runtime; handy for spotting stuck long jobs.

By user / name

```
ps -u www-data -o pid,cmd
ps -C nginx -o pid,cmd
pgrep -a nginx          # often clearer for name → PID
```

Tree view

```
ps auxf | less
ps -ejH | less
pstree -ap | less
```

Shows parent/child relationships (workers under a master, shells under `sshd`).

One PID in detail

```
ps -p 1 -o pid,user,cmd,lstart,etime
tr '\0' ' ' < /proc/1/cmdline; echo
```

/proc/PID is the source of truth when `ps` columns are not enough.

Threads

```
ps -eLf | head
ps -p "$(pgrep -n nginx)" -L -o pid,tid,pcpu,cmd
```

`-L / lwp` shows threads; high thread counts can matter for ulimits.

Full command line (no truncation)

```
ps -eo pid,args | less
ps auxww | less           # wide; ww avoids early wrap on BSD style
```

Default width may truncate CMD; widen the terminal or use `args/ww`.

Zombies and D-state

```
ps aux | awk '$8 ~ /Z/ {print}'      # zombies (STAT has Z)
ps aux | awk '$8 ~ /D/ {print}'      # uninterruptible sleep (often I/O)
```

Zombies need parent reaping; D state often means storage/NFS pain.

Script: fail if process missing

```
ps -C mydaemon >/dev/null || { echo "mydaemon not running" >&2; exit 1; }
pgrep -x mydaemon >/dev/null
```

Understanding Output

- **STAT**: R running, S interruptible sleep, D uninterruptible, Z zombie, T stopped; extras like + (foreground), l (multi-threaded), < (high priority).
- **%CPU**: not identical to `top`'s averaged view.

- **VSZ vs RSS:** virtual size vs resident RAM.
- **USER:** effective user; real UID can differ for setuid binaries.

Notes & Pitfalls

- `ps` is a **point in time**; loops need `watch` or a real monitor.
- Prefer `pgrep/pkill` over `ps | grep` (avoids matching the `grep` itself).
- Kernel threads appear as `[kthreadd]`-style names.
- Container views may only show processes in the same PID namespace.

Related Commands

- `top` / `htop` / `btm` — interactive
- `pgrep` / `pkill` — select/signal by pattern
- `ps` — hierarchy
- `pidof` — simple name lookup
- `/proc/PID` — kernel process API

Additional Resources

- `man ps`
- `man proc`

pstree

Overview

pstree shows running processes as a **tree**, making parent/child relationships obvious — ideal for seeing what a supervisor spawned, which shell owns a pipeline, or how threads hang under a process.

Syntax

```
pstree [options] [pid|user]
```

Common Options

Option	Description
-a	Show command-line args
-p	Show PIDs
-c	Don't compact identical subtrees
-h / -H pid	Highlight current / specified
-n	Sort by PID
-u	Show uid transitions
-g	Show PGID
-s	Show parents of specified PID
-l	Long lines (no truncation)
-T	Hide threads
-A / -U	ASCII / UTF-8 lines
-Z	SELinux context
-t	Show full thread names when available

Examples with Explanations

Basics

```
pstree
pstree -p
pstree -ap
```

One user or PID

```
pstree alice
pstree -p 1
pstree -ap $(pidof -s nginx)
```

Show ancestry of a process

```
pstree -s -p 12345
ps -o pid,ppid,cmd -p 12345
```

Avoid compaction

```
pstree -c -p
```

Identical child names are otherwise merged with counts.

Threads vs processes

```
pstree -p 12345
pstree -T -p 12345
```

SELinux

```
pstree -Z
```

Compare with ps

```
ps auxf
pstree -ap
systemd-cgls
```

`ps f` / `ps --forest` is another forest view; `systemd-cgls` shows cgroup trees.

Notes / Pitfalls

- Large trees on busy hosts are noisy — filter by user/PID.
- Compact mode can hide multiplicity — use `-c` when counting workers.
- Permission: some args may be invisible for other users' processes.
- PID namespaces in containers show a nested world (PID 1 is container init).
- Not a real-time monitor — re-run or use `watch -n1 pstree -p`.

2026-relevant notes

- systemd-heavy systems: also learn `systemctl status` and `systemd-cgtop`.
- For containers: `podman top` / `docker top` + host `pstree -p` with care.
- Kernel threads appear under `kthreadd` (PID 2) on the host.

Related Commands

- `ps auxf` — forest view
- `pgrep` / `pidof` — find PIDs
- `systemd-cgls` — cgroup tree
- `top` / `htop` — live views with tree modes
- `pstree` package often from `psmisc`

Additional Resources

- `man pstree`

top

Overview

`top` is the classic interactive process monitor: CPU, memory, and a live-sorted process table. It is installed almost everywhere, so it is the default triage tool when `htop`/`btop` are not present. For scripts and snapshots, prefer `ps`; for friendlier interactive use, install `htop`.

Syntax

```
top [options]
```

Common Options

Option	Description
<code>-d SECS</code>	Delay between refreshes (fractional OK)
<code>-n COUNT</code>	Exit after COUNT iterations (batch-friendly)
<code>-b</code>	Batch mode (non-interactive; good for logs)
<code>-p PID, -p PID1,PID2</code>	Watch only these PIDs
<code>-u USER / -U USER</code>	Filter by effective / any user id
<code>-H</code>	Show threads
<code>-o FIELD</code>	Sort by field (e.g. %CPU, %MEM)
<code>-c</code>	Toggle command line vs name (often available as interactive <code>c</code>)
<code>-w</code>	Wide output (batch)

Interactive keys (procps-ng)

Key	Action
<code>h / ?</code>	Help
<code>q</code>	Quit
<code>Space / Enter</code>	Refresh now
<code>d / s</code>	Change delay
<code>P</code>	Sort by CPU

Key	Action
M	Sort by memory
T	Sort by time
N	Sort by PID
k	Kill (prompt for PID and signal — TERM default)
r	Renice
u	Filter user
c	Command line vs basename
V	Forest / tree view (when supported)
H	Toggle threads
1	Expand / collapse per-CPU lines
f / F	Field management
W	Write config (~/.config/procps/toprc or ~/.toprc)

Key Use Cases

1. Live “who is burning CPU/RAM right now”
2. Quick renice or kill during an incident
3. Capture a few samples in batch for a ticket
4. Per-CPU and load-average glance during overload

Examples with Explanations

Example: interactive default

```
top
```

Header shows uptime, load average, task counts, CPU breakdown, and memory. Process list is typically sorted by %CPU.

Example: refresh every two seconds

```
top -d 2
```

Slower refresh reduces overhead on already-loaded systems.

Example: single-user focus

```
top -u www-data
```

Limits noise on multi-tenant or app servers.

Example: watch specific PIDs

```
top -p 1,1234,$(pgrep -d, -x nginx)
```

Useful when you already know the suspects from `ps/pgrep`.

Example: batch sample for logs / tickets

```
top -b -n 3 -d 1 > /tmp/top-snapshot.txt  
top -b -n 1 -o %MEM | head -n 30
```

`-b` makes output non-interactive and redirectable. Great for attaching evidence to an incident note.

Example: sort by memory from the CLI

```
top -o %MEM
```

Start already sorted by resident memory pressure.

Example: threads view

```
top -H -p "$(pgrep -d, -x java)"
```

Each thread appears as a row — helpful for multi-threaded runtimes.

Example: kill from inside top

Press `k`, enter PID, accept default signal **15 (TERM)** unless you have already tried graceful stop. Escalate to 9 only if needed.

Example: one-shot CPU leaders without staying in top

```
ps aux --sort=-%cpu | head -n 15
```

When you only need a snapshot, `ps` is lighter and script-friendlier.

Understanding Output

Header

- **load average** — 1/5/15 minute runnable+uninterruptible averages (not “percent CPU”)
- **Cpu(s)** — %**us** user, %**sy** system, %**id** idle, %**wa** I/O wait, %**st** steal (VMs)
- **MiB Mem / Swap** — total, free, used, buff/cache (wording varies by procp version)

Process columns (typical)

Column	Meaning
PID	Process ID
USER	Effective user
PR / NI	Priority / nice
VIRT / RES / SHR	Virtual / resident / shared memory
S	State (R run, S sleep, D disk sleep, Z zombie, T stopped)
%CPU / %MEM	Share of CPU / physical memory
TIME+	Cumulative CPU time
COMMAND	Name or full command

On multi-core hosts, %CPU can exceed 100% for multi-threaded processes.

Notes & Pitfalls

- **top** is **live**; do not parse interactive mode in scripts — use **-b** or **ps**.
- High %**wa** points to storage/I/O, not always “need more CPU”.
- High %**st** (steal) means the hypervisor is scheduling other VMs — look outside the guest.
- Memory: **buff/cache** is usually reclaimable; “free” alone is a poor metric — check available if shown.
- Killing from **top** still needs the same discipline as **kill**: **TERM** before **KILL**.
- Config saved with **W** can surprise you later if sort/fields change.

Related Commands

- **htop** / **bttop** — modern interactive monitors
- **ps** — snapshots and scripting
- **uptime** / **vmstat** / **mpstat** — load and CPU breakdown
- **pidstat** — per-process rates over time
- **free** — memory summary

Additional Resources

- `man top`
- `man procps`

htop

Overview

htop is an interactive process viewer with a clearer UI than classic **top**: mouse support, tree view, easy search/filter, color meters, and safer multi-select kill. It is **not** always installed by default on Ubuntu servers — install with **apt**. For automation, keep using **ps/pgrep**; **htop** is for humans at a terminal.

Syntax

```
htop [options]
```

Common Options

Option	Description
-d DELAY	Update delay in tenths of a second (-d 10 1s)
-u USER	Show only this user
-p PID[,PID...]	Restrict to these PIDs
-t, --tree	Start in tree view
-C, --no-color	Monochrome
-s COLUMN	Sort by column name
-H DEPTH	Hide threads beyond depth (version-dependent)

Install on Ubuntu:

```
sudo apt update
sudo apt install htop
```

Useful keys

Key	Action
F1 / h	Help

Key	Action
F2 / S	Setup (meters, columns, display options)
F3 / /	Search
F4 / \	Filter (hide non-matches)
F5 / t	Tree view
F6 / >	Choose sort column
F7 / F8	Decrease / increase nice
F9 / k	Kill — pick signal from a list (prefer TERM)
Space	Tag process
c	Tag process and its children (with tree)
U	Untag all
u	User filter
H	Hide user threads
K	Hide kernel threads
M / P / T	Sort memory / CPU / time
p	Show full paths
I	Invert sort
l	List open files (lsof) if available
s	strace selected process (if permitted)
q	Quit

Key Use Cases

1. Interactive incident triage with filters and trees
2. Finding the parent of a worker storm
3. Niceness tweaks and careful multi-process kill
4. Per-user resource glance on shared hosts

Examples with Explanations

Example: start htop

```
htop
```

Meters at the top, process table below. Use mouse or keys to select rows.

Example: only your processes

```
htop -u "$USER"
```

Cuts system noise during personal debugging.

Example: tree view for process storms

```
htop -t  
# or press F5 inside htop
```

Parent/child relationships show which master spawned runaway workers.

Example: attach to known PIDs

```
htop -p "$(pgrep -d, nginx)"
```

Focuses the display on nginx-related PIDs only.

Example: slower refresh on overloaded boxes

```
htop -d 20
```

Two-second delay (20 tenths) reduces sampling cost while you think.

Example: kill workflow inside htop

1. Filter with F4 or search with F3
2. Tag targets with Space (optional)
3. F9 / k → choose **SIGTERM** first
4. Wait; only then send **SIGKILL** to leftovers

Never default to SIGKILL out of habit.

Example: nice a batch job interactively

Select the process → F8 a few times to make it friendlier (higher nice), or F7 if you have privilege to make it more aggressive.

Example: setup persistence

F2 → adjust CPU meter style, add columns (IO_RATE, STARTTIME, ...) → leave setup; htop writes ~/.config/htop/htoprc automatically on exit in modern versions.

Understanding Output

- **CPU bars** — per-core usage; colors distinguish user/system/irq/nice/io-wait/steal depending on version and setup.
- **Memory bar** — used vs buffers/cache; green/blue conventions differ from **free** wording but convey pressure.
- **Load average** — same kernel figures as **uptime/top**.
- **Process states** — same single-letter codes as **ps** (R, S, D, Z, T, ...).

Notes & Pitfalls

- Not for scripts: output is interactive and not a stable API.
- Kill and renice still obey **permissions** — root for others' processes and for negative nice.
- Kernel threads clutter the view; toggle K/H for clarity.
- Container hosts: PIDs may be host-namespaced depending on where you run **htop**.
- Alternatives: **btop** (pretty), **atop** (historical accounting), classic **top** (always there).

Related Commands

- **top** — ubiquitous interactive monitor
- **btop** — modern TUI alternative
- **ps** / **pgrep** — snapshots and selection
- **vmstat** / **iostat** / **pidstat** — subsystem rates
- **nice** / **renice** / **ionice** — priority controls

Additional Resources

- `man htop`
- htop.dev

procs

Overview

`procs` is a modern replacement for `ps` written in Rust, with colorized output, tree views, keyword highlighting, and friendlier defaults. Optional install (`cargo install procs`, distro packages). Keep `ps` for portability and scripts.

Syntax

```
procs [options] [query...]  
procs --help
```

Common Options

Option	Description
<code>-a, --and</code>	All keywords must match
<code>-o, --or</code>	Any keyword (default-ish behavior depends version)
<code>-d, --tree</code>	Tree view
<code>-w, --watch</code>	Watch mode refresh
<code>-p, --pager</code>	Use pager
<code>--sortd KEY / --sorta KEY</code>	Sort descending/ascending
<code>--only COL</code>	Show specific columns
<code>--insert COL</code>	Add columns
<code>--json</code>	JSON output
<code>--color</code>	Color mode
<code>query</code>	Filter by keyword (command, user, ...)

Flags evolve quickly — check `procs --help` for your version.

Examples with Explanations

Everyday

```
procs  
procs nginx  
procs alice  
procs --tree  
procs -w
```

Sorting

```
procs --sortd cpu  
procs --sortd mem  
procs --sorta pid
```

Columns

```
procs --only pid,user,cmd  
procs --insert tcp,udp
```

JSON for tooling

```
procs --json nginx | jq .
```

Compare with ps

```
ps aux | head  
procs  
ps -fp $(pgrep -d, nginx)  
procs nginx
```

Watch like top (lightweight)

```
procs -w  
# or  
watch -n1 procs --sortd cpu
```

Notes / Pitfalls

- Not installed by default on servers — don't require it in production scripts.
- Keyword filtering is convenience, not a security boundary.
- Column names differ from `ps` — learn the map once.
- JSON schema may change across versions; pin version in automation.
- Permissions still limit visibility of other users' full command lines.

2026-relevant notes

- Part of the modern CLI set with `bat`, `eza`, `fd`, `rg`, `bt`.
- For portable ops docs, show `ps` equivalents alongside `procs`.
- Prefer `pgrep/systemctl` for scripting; use `procs` interactively.

Related Commands

- `ps` — portable process status
- `pgrep` / `pkill` — match / signal
- `top` / `htop` / `bt` — interactive monitors
- `pstree` — tree view
- `pidof` — PIDs by name

Additional Resources

- `procs --help`
- [dalance/procs](#)

pgrep

Overview

`pgrep` lists process IDs matching selection criteria (name, full command line, user, terminal, etc.). `pkill` uses the **same selectors** but sends a signal instead of printing PIDs. Together they replace brittle `ps | grep` pipelines for operations and scripts. On Ubuntu both come from **procps**.

Syntax

```
pgrep [options] pattern
pkill [options] pattern
```

Common Options

Shared selection flags (both tools):

Option	Description
<code>-u USER, --uid</code>	Match effective user (name or UID)
<code>-U USER, --euid</code>	Match real user (see man for distinction)
<code>-g PGID / -G GID</code>	Process group / group membership
<code>-t TERM</code>	Controlling terminal (e.g. <code>pts/0</code>)
<code>-P PPID</code>	Children of PPID
<code>-f, --full</code>	Match against full command line, not only comm
<code>-x, --exact</code>	Exact match of the name (or full line with <code>-f</code>)
<code>-n / -o</code>	Newest / oldest match only
<code>-v</code>	Inverse match
<code>-c</code>	Count matches (<code>pgrep</code>)
<code>-d DELIM</code>	PID delimiter (default newline; <code>,</code> useful for <code>htop -p</code>)
<code>-a</code>	Show PID and full command (<code>pgrep</code>)
<code>-l</code>	Show PID and process name (<code>pgrep</code>)
<code>-i</code>	Case-insensitive
<code>-r</code>	Ignore descendants of <code>pgrep/pkill</code> itself (where supported)

`pkill`-specific:

Option	Description
<code>-SIGNAL / -s SIGNAL</code>	Signal name or number (default TERM)
<code>-e, --echo</code>	Print what will be signaled (newer procs)
<code>-q</code>	Quiet

Safety

- Broad patterns (`kill python`, `kill java`) are classic outage sources — prefer **-x**, **-f** with a specific string, and **-u**.
- Escalate signals: **TERM** → **wait** → **KILL**; `kill -9` should be rare.
- Managed services: prefer `systemctl stop/reload` over `kill nginx`.
- Always inventory with `pgrep -a` before `kill` on production.

Key Use Cases

1. Resolve PIDs for scripts without parsing `ps`
2. Signal a homogeneous worker pool
3. Count processes for simple health checks
4. Feed PID lists to `top/htop/gdb`

Examples with Explanations

Example: list matching PIDs

```
pgrep sshd
pgrep -a nginx
pgrep -l python3
```

`-a` shows full command lines — best pre-kill inspection.

Example: exact name match

```
pgrep -x sshd
pgrep -x nginx
```

Avoids partial matches against longer comm names when you want precision.

Example: full command line

```
pgrep -af 'gunicorn.*myapp'  
pgrep -f 'python3 /opt/app/worker.py'
```

-f is essential when every process is python3 or java.

Example: filter by user

```
pgrep -u www-data -a php-fpm  
pgrep -u "$USER" -f train.py
```

Reduces collateral damage on shared systems.

Example: count for monitoring

```
pgrep -c -x sidekiq  
pgrep -c -u redis redis-server
```

Exit status is still 0/1 for found/not found; combine carefully in scripts.

Example: newest / oldest only

```
pgrep -n -x myapp      # newest  
pgrep -o -x myapp      # oldest (often the master)
```

Useful when a master/worker model embeds the same name.

Example: children of a parent

```
pgrep -P "$(pgrep -xo master-process)"
```

List direct children by parent PID.

Example: delimiter for other tools

```
htop -p "$(pgrep -d, nginx)"  
top -p "$(pgrep -d, -x myapp)"
```

Comma-separated PID lists fit `htop -p` / `top -p`.

Example: pkill graceful then force

```
pkill -TERM -x myapp  
sleep 3  
pkill -KILL -x myapp 2>/dev/null
```

Same discipline as `kill`: `TERM` first, `KILL` only for stragglers.

Example: pkill with full-line match

```
pkill -u deploy -f 'unicorn app.main:app'
```

Targets one service invocation pattern under one user.

Example: avoid ps | grep

```
# fragile:  
ps aux | grep '[n]ginx'  
# better:  
pgrep -a nginx
```

Character-class `grep` tricks are obsolete for this job.

Understanding Output

- Default: one PID per line.
- Exit status **0** if any match, **1** if none, **2** on syntax/usage error, **3** fatal error (procp's conventions).
- `pkill` exit status follows the same “matched or not” idea — check man page when scripting strict failure modes.

Notes & Pitfalls

- Pattern is an **extended regex** by default — dots and brackets matter; quote patterns.
- Without `-f`, matching uses the process name (comm), often truncated to 15 characters historically — use `-f` for long names.
- Race: process can exit between `pgrep` and `kill`; handle missing PIDs.
- `pkill/pgrep` may see processes in other sessions; still restricted by permissions for signaling.
- In containers, you only see PIDs in your PID namespace.

Related Commands

- `pkill` — signal instead of list (same selectors)
- `kill` / `killall` — by PID or simple name
- `pidof` — simpler name → PID (fewer filters)
- `ps` — rich table and custom columns
- `systemctl` — prefer for supervised units

Additional Resources

- `man pgrep`
- `man pkill`

pidof

Overview

`pidof` prints the PIDs of running programs matched by name. It is a quick alternative to `pgrep -x` for simple lookups and remains common in init scripts and one-liners. For flexible pattern matching, prefer `pgrep`.

Syntax

```
pidof [options] program [program...]
```

Common Options

Option	Description
<code>-s</code>	Single PID only (first/one)
<code>-c</code>	Only processes with same root (chroot)
<code>-x</code>	Also match scripts (by script name)
<code>-o omitpid</code>	Omit PID(s); often <code>-o %PPID</code> or self
<code>-z</code>	Skip zombies (when supported)

Options vary slightly between implementations (sysvinit-utils vs others).

Examples with Explanations

Basics

```
pidof nginx
pidof sshd
pidof bash
```

Single PID

```
pidof -s nginx
kill -HUP "$(pidof -s nginx)"
```

Scripts

```
pidof -x backup.sh
pgrep -f backup.sh          # often clearer for scripts
```

Existence gate

```
if pidof dockerd >/dev/null; then
    echo docker running
fi
```

Omit PIDs

```
pidof -o $$ bash          # other bashes, not this script's shell if named bash
pidof -o %PPID nginx
```

Feed to kill

```
kill $(pidof myapp)
# safer with pgrep:
pgrep -x myapp | xargs -r kill
```

Compare tools

```
pidof nginx
pgrep -x nginx
pgrep -a nginx
systemctl status nginx
```

Notes / Pitfalls

- Name matching is not always exact across threads/workers — verify with `ps`.
- Multiple PIDs print space-separated — quote carefully in scripts.
- Race: process may exit between `pidof` and `kill`.
- Containers: only sees processes in the same PID namespace.
- Prefer `systemctl` for supervised services instead of raw PID kills.

2026-relevant notes

- `systemd` units: `systemctl show -p MainPID nginx`.
- `pgrep/pkill` from `procps` are more expressive for modern scripts.
- Rootless stacks may have multiple same-named processes per user — be specific.

Related Commands

- `pgrep` / `pkill` — pattern match / signal
- `kill` / `killall` — signal by PID / name
- `ps` — detailed process list
- `systemctl` — service PIDs
- `pidwait` / `pwait` — wait for PID exit (if installed)

Additional Resources

- `man pidof`

kill

Overview

kill sends a signal to one or more processes by **PID**. Despite the name, the default signal is **TERM (15)** — a polite request to exit. Only escalate to **KILL (9)** after a graceful stop fails. Prefer `systemctl stop` / `systemctl reload` for units managed by systemd.

Syntax

```
kill [options] PID...
kill -SIGNAL PID...
kill -s SIGNAL PID...
kill -l [signal]
```

Common Options

Option	Description
-s SIGNAL, -SIGNAL	Signal name or number (TERM, HUP, 9, ...)
-l	List signal names (or translate a number → name)
-n SIGNAL	Same idea as -s on some implementations
-p (bash builtin only)	Print PID instead of signaling (shell-specific)

Signals you will use constantly:

Signal	Number	Typical meaning
TERM	15	Graceful stop (default)
HUP	1	Hangup / often “reload config”
INT	2	Interrupt (like Ctrl-C)
QUIT	3	Quit with core (if allowed)
KILL	9	Force kill — uncatchable
STOP	19	Pause (uncatchable)
CONT	18	Resume after STOP
USR1 / USR2	10 / 12	Application-defined

Safety

1. **Always verify the PID** before signaling: `ps -p PID -o pid,user,cmd` or `ps -fp PID`.
2. **TERM first, then wait, then KILL** — never jump to `-9` on production workers unless the process is wedged.
3. Wrong PID on a multi-tenant host can take down databases, SSH, or the wrong customer job.
4. You may only signal processes you own unless you are root.
5. Prefer **service managers** for supervised daemons: `systemctl stop UNIT`, not ad-hoc `kill` on random PIDs from `ps`.

Key Use Cases

1. Stop a runaway user process after verifying the PID
2. Reload a daemon that documents `SIGHUP` handling
3. Force-kill a process stuck in userspace after `TERM` fails
4. Test whether a PID still exists (`kill -0`)

Examples with Explanations

Example: graceful stop (default TERM)

```
ps -fp 1234
kill 1234
# same as:
kill -TERM 1234
kill -15 1234
kill -s TERM 1234
```

Default signal is `TERM`. Confirm the process is the one you intend with `ps` first.

Example: TERM, wait, then KILL

```
PID=1234
kill -TERM "$PID"
for i in 1 2 3 4 5; do
    kill -0 "$PID" 2>/dev/null || break
    sleep 1
done
if kill -0 "$PID" 2>/dev/null; then
    echo "still alive; forcing" >&2
    kill -KILL "$PID"
fi
```


`kill -0 PID` checks existence and permission **without** delivering a real signal. Use this pattern instead of blind `kill -9`.

Example: force kill only when necessary

```
kill -KILL 1234
# or
kill -9 1234
```

KILL cannot be caught or cleaned up. File locks, temp files, and child reaping may be messy. Prefer TERM whenever the process still responds.

Example: reload via HUP

```
kill -HUP "$(pidof nginx)"
# better for packaged services:
sudo systemctl reload nginx
```

Many classic daemons reload config on HUP. Prefer `systemctl reload` so systemd tracks state and uses the unit's configured reload command.

Example: signal by name via pgrep

```
kill $(pgrep -x myapp)
# often clearer:
pkill -x myapp
pkill -TERM -x myapp
```

Shell word-splitting on `$(pgrep ...)` is fine for PIDs; still prefer exact match (`-x`) so you do not hit similarly named processes.

Example: stop / continue (debug freeze)

```
kill -STOP 1234    # freeze
# inspect, attach debugger, etc.
kill -CONT 1234    # resume
```

STOP/CONT are useful for debugging or temporarily freezing a CPU hog. A STOP'd process will not exit until CONT (or KILL).

Example: list signals

```
kill -l
kill -l 15      # → TERM (often)
kill -l TERM    # → 15
```

Handy when docs show only numbers or only names.

Example: multiple PIDs

```
kill -TERM 1001 1002 1003
kill -TERM $(pgrep -u "$USER" -f 'worker.py')
```

One invocation can target many PIDs. Combine with careful `pgrep` filters (`-u`, `-f`, `-x`).

Example: permission check without killing

```
if kill -0 1234 2>/dev/null; then
    echo "process exists and is signalable"
else
    echo "gone or not permitted"
fi
```

Exit status of `kill -0` is ideal for scripts that poll for death after `TERM`.

Understanding Output

- Success is silent; failures print to `stderr` (No such process, Operation not permitted).
- Exit status non-zero if **any** PID failed (implementation details vary slightly for multi-PID).
- Bash has a `kill` **builtin** that may differ slightly from `/bin/kill` (util-linux); both understand standard signals.

Notes & Pitfalls

- **Zombies** (Z state) ignore signals; kill the **parent** or fix whatever is not `wait()`ing.
- Processes in uninterruptible sleep (D state, often I/O) may not die until the wait ends — `KILL` does not magically fix stuck NFS or bad block devices.
- Killing PID **1** is a terrible idea; on systemd hosts it will refuse nonsense, but do not experiment.
- Job-control shells also understand `%1` job specs with the **builtin** `kill` — different from numeric PIDs.
- Containers: killing PID 1 inside a container usually stops the container; know your runtime's restart policy.

Related Commands

- `pkill` / `pgrep` — select and signal by name/pattern
- `killall` — signal by process name (GNU vs BSD differences)
- `systemctl` — stop/reload supervised services
- `timeout` — auto-signal a command after a duration
- `ps` / `pidof` — identify targets
- `fuser` — signal processes holding files or ports

Additional Resources

- `man 1 kill`
- `man 7 signal`

killall

Overview

`killall` sends a signal to all processes matching a **command name**. On Ubuntu it is the **psmisc** implementation (name match), not the ancient SysV “kill everything” binary. Prefer **pkill** when you need regex/full-command-line filters; use **killall** for simple exact-name mass signals. Prefer **systemctl** for managed services.

Syntax

```
killall [options] name...
```

Common Options

Option	Description
<code>-s SIGNAL, -SIGNAL</code>	Signal to send (default TERM)
<code>-e, --exact</code>	Require exact name match (long names)
<code>-I, --ignore-case</code>	Case-insensitive name
<code>-i, --interactive</code>	Prompt before each kill
<code>-l, --list</code>	List known signal names
<code>-u USER, --user</code>	Only this user's processes
<code>-v, --verbose</code>	Report what was signaled
<code>-w, --wait</code>	Wait until processes die
<code>-r, --regex</code>	Interpret name as extended regex (psmisc)
<code>-n, --ns PID</code>	Match processes in same namespaces as PID
<code>-Z, --context</code>	SELinux context match (when relevant)
<code>-q, --quiet</code>	Suppress “no process found” complaint

Safety

- Name matching can still hit **more instances** than you expect (every **python3**, every **java**).
- Prefer **TERM first**; only use **-9** after a wait.
- Always narrow with **-u** on shared hosts.
- Interactive confirm (**-i**) is cheap insurance on production bastions.

- Do not confuse with Solaris/AIX/older docs where `killall` meant something more aggressive.

Key Use Cases

1. Stop all instances of a user-space tool by binary name
2. Reload or terminate workers that are not under systemd
3. Interactive cleanup of leftover processes after a failed deploy
4. Wait until a named process set has exited before continuing a script

Examples with Explanations

Example: graceful stop by name

```
killall myapp
killall -TERM myapp
```

Sends `TERM` to every process whose `comm`/name matches `myapp`. Equivalent intent to `pkill -x myapp` in many cases.

Example: `TERM` then `KILL`

```
killall -TERM myapp
sleep 3
killall -KILL myapp 2>/dev/null
```

Give the app time to flush and exit; force only stragglers. Redirect `stderr` if absence after `TERM` is expected success.

Example: only your processes

```
killall -u "$USER" -TERM chrome
killall -u deploy -TERM 'node'
```

Critical on multi-user machines so you do not signal another user's same-named binary.

Example: interactive confirm

```
killall -i -TERM python3
```

Prompts per process — use when several interpreters are running and only some should die.

Example: verbose + wait

```
killall -v -w myapp
echo "all myapp processes gone"
```

`-w` blocks until matching processes exit (or cannot be killed). Useful in stop scripts before releasing a lock or port.

Example: HUP reload pattern

```
killall -HUP nginx
# preferred on Ubuntu for packaged nginx:
sudo systemctl reload nginx
```

HUP is conventional for reload; systemd still owns the proper lifecycle for units.

Example: case-insensitive / regex

```
killall -I MyApp
killall -r 'worker-[0-9]+'
```

`-r` uses extended regular expressions (psmisc). Test with `pgrep` first when unsure.

Example: list signals

```
killall -l
```

Same idea as `kill -l`.

Example: dry reconnaissance first

```
pgrep -a myapp
ps -C myapp -o pid,user,cmd
killall -i -TERM myapp
```

Always inventory before mass signaling when the blast radius is unclear.

Notes & Pitfalls

- **GNU/psmisc vs BSD:** FreeBSD `killall` options differ — scripts should stick to portable flags or use `pkill`.
- Matching is by process **name** (comm), not necessarily full path; long names may need `-e`.
- `killall python` will not match `python3` unless you name it correctly (or use regex).
- Exit status is non-zero when **no** process matched (unless quieted) — check that in scripts.
- Kernel threads and some system processes cannot be killed; errors are expected.
- Prefer `systemctl stop foo` over `killall foo` for daemons shipped as units.

Related Commands

- `kill` — signal by PID
- `pkill` / `pgrep` — pattern-based select/signal
- `pidof` — simple name → PID list
- `systemctl` — supervised stop/reload
- `kill` / `pkill` — more controlled signaling on servers

Additional Resources

- `man killall` (psmisc)
- `man 7 signal`

jobs

Overview

`jobs` lists **active jobs** of the current shell: running background tasks, stopped jobs, and their job IDs. Essential for job control with `fg`, `bg`, `kill %n`, and `wait`.

Jobs are not a system-wide process table — use `ps/pgrep` for that.

Syntax

```
jobs [options] [job_spec ...]
```

Common Options (bash)

Option	Description
<code>-l</code>	Include PIDs
<code>-p</code>	PIDs only
<code>-n</code>	Only jobs with status changes since last notification
<code>-r</code>	Running only
<code>-s</code>	Stopped only
<code>-x command</code>	Replace job specs in command with PIDs and run

Job states

State	Meaning
Running	Executing in background
Stopped	Suspended (Ctrl-Z / SIGSTOP)
Done	Completed successfully
Exit N	Exited with status N
Terminated	Killed by signal

Examples with Explanations

List jobs

```
sleep 300 &  
sleep 400 &  
jobs  
jobs -l  
jobs -p  
jobs -r  
jobs -s
```

Act on a job

```
jobs -l  
fg %1  
bg %2  
kill %1  
kill -9 %2
```

Status changes

```
jobs -n
```

Useful after many completions to see what finished.

wait integration

```
longtask &  
pid=$!  
jobs -l  
wait "$pid"
```

Enable job control in scripts (rare)

```
set -m  
sleep 5 &  
jobs -l  
wait
```

disown removes from table

```
sleep 1000 &
jobs
disown
jobs          # empty regarding that job
# process may still run - check ps
```

Notes / Pitfalls

- Empty `jobs` does not mean no processes exist — only no **shell jobs**.
- Job IDs are not PIDs; use `jobs -l` or `$!` for PIDs.
- Subshells have their own job tables: (`sleep 10 & jobs`).
- Non-interactive shells often disable job control until `set -m`.
- Don't use `jobs` for service supervision — use `systemd`.

2026-relevant notes

- Interactive terminal multiplexers reduced reliance on many concurrent jobs, but `%` job specs remain daily muscle memory.
- CI shells may not support job control the same way — use PIDs + `wait`.
- For parallel shell work, GNU `parallel` or `xargs -P` may be clearer than raw jobs.

Related Commands

- `fg` / `bg` — resume jobs
- `kill %n` — signal jobs
- `wait` — wait for completion
- `disown` — remove from job table
- `ps` / `pgrep` — system-wide process view
- `set -m` — monitor mode

Additional Resources

- `help jobs`, `man bash`

fg

Overview

fg resumes a job in the **foreground**, giving it the terminal again (stdin/stdout/signals). Use it to continue a stopped job interactively or to pull a background job back for interaction.

Syntax

```
fg [job_spec]
```

Default is the current job (%+).

Job specification

Spec	Meaning
%n	Job number n
%name	Command begins with name
%?str	Command contains str
%% / %+	Current
%-	Previous

Examples with Explanations

After Ctrl-Z

```
vim bigfile.txt
# Ctrl-Z
jobs
fg
# or
fg %vim
```

Classic suspend-and-resume editor workflow.

Pull background job forward

```
./server &  
jobs -l  
fg %1  
# Ctrl-C now kills the server (foreground signals)
```

Switch between jobs

```
jobs  
fg %1  
# Ctrl-Z  
fg %2
```

With pipelines

```
tar czf - dir | ssh host 'cat > backup.tgz' &  
jobs  
fg %1
```

Script note

```
set -m  
sleep 5 &  
fg %1
```

Usually unnecessary — scripts should `wait` on PIDs instead.

Notes / Pitfalls

- Only jobs of the current shell.
- Foreground job receives terminal-generated signals (`Ctrl-C` → `SIGINT`, `Ctrl-Z` → `SIGTSTP`).
- If job is already foreground, `fg` errors.
- Background jobs needing stdin will stop with `SIGTTIN` — `fg` to provide input.
- Job numbers change as jobs complete — verify with `jobs` first.

2026-relevant notes

- Interactive recovery still common; long-running prod work belongs in systemd/tmux.
- SSH drop kills foreground jobs; use multiplexers for remote interactive work.
- Know difference: `fg` (shell job) vs `docker attach` / `kubectl attach` (container).

Related Commands

- `bg` — resume in background
- `jobs` — list
- `kill %1` — signal a job
- `Ctrl-Z` — stop foreground
- `tmux` — better multi-tasking than many suspended jobs

Additional Resources

- `help fg`, `man bash`

bg

Overview

bg resumes a **stopped** job in the **background** so it continues running without occupying the terminal. It is part of interactive shell **job control** (bash/zsh), together with **fg**, **jobs**, **Ctrl-Z**, and **&**.

Job control is per-shell: background jobs are not the same as systemd services or **nohup** daemons.

Syntax

```
bg [job_spec ...]
```

With no argument, resumes the current job (%+).

Job specification

Spec	Meaning
%n	Job number n
%string	Job whose command begins with string
%?string	Job whose command contains string
%% or %+	Current job
%-	Previous job

Common Options

None significant — **bg** is a shell builtin. See **help bg**.

Examples with Explanations

Suspend then background

```
# start a long command in foreground
find / -xdev -name '*.log' 2>/dev/null
# press Ctrl-Z → Stopped
bg
# continues in background
jobs -l
```

Start in background directly

```
make -j$(nproc) &
jobs
```

& starts background immediately; bg is for jobs already stopped.

Specific job

```
jobs
bg %1
bg %find
```

Multiple jobs

```
bg %1 %2
```

Typical workflow

```
gzip -k huge.img
# Ctrl-Z
bg
# continue using the shell
jobs -l
fg %1          # bring back if needed
```

Disown / survive hangup

```
# after bg:
disown %1
# or start with:
nohup longcmd &
# or use tmux/systemd for real persistence
```

Not for scripts by default

```
# non-interactive shells often have job control off
set -m          # monitor mode (enable job control) if needed
sleep 30 &
jobs
```

Prefer explicit background with `&` and `wait` in scripts rather than `bg`.

Notes / Pitfalls

- Only works with **job control enabled** and jobs owned by this shell.
- Background jobs writing to the terminal can interleave messily — redirect stdout/stderr.
- `bg` on an already running background job is a no-op / error depending on shell.
- Closing the terminal may SIGHUP jobs unless `disown`/`nohup`/`tmux`/`systemd`.
- Don't confuse with `systemd bg` — there is none; use units for services.

2026-relevant notes

- For anything important, prefer `tmux/systemd-run/nohup` over bare `bg`.
- Desktop environments may use their own session management; servers still rely on classic job control daily.
- Pipeline job control can be surprising — know whether the whole pipeline is one job.

Related Commands

- `fg` — resume in foreground
- `jobs` — list jobs
- `Ctrl-Z` / `kill -STOP` — stop
- `disown` — detach from shell job table
- `nohup` — ignore hangup
- `tmux` / `screen` — persistent sessions
- `wait` — wait for background PIDs/jobs

Additional Resources

- `help bg`, `man bash` (JOB CONTROL)

nohup

Overview

`nohup` runs a command immune to **SIGHUP**, so it keeps running after you log out of a terminal session. It is a minimal “survive hangup” wrapper — not a full service manager. For interactive long-lived SSH work prefer **tmux/screen**; for production daemons prefer **systemd** (user or system units).

Syntax

```
nohup COMMAND [ARGS] ...
nohup COMMAND [ARGS] ... &
```

Common Options

Option	Description
<code>--help</code>	Usage
<code>--version</code>	Version

`nohup` itself has almost no flags; behavior is about signals, redirection, and the shell background operator `&`.

Behavior (what you actually care about)

- Ignores **SIGHUP** for the started command.
- If **stdout** is a **terminal**, output is appended to `./nohup.out` (or `$HOME/nohup.out` if the cwd is not writable).
- If you already redirect stdout/stderr, `nohup` does not create `nohup.out`.
- Exit status is that of **COMMAND**, or 127 if the command cannot be invoked.

Key Use Cases

1. Kick off a one-off long job before disconnecting SSH (when you did not start tmux)
2. Quick background batch with a simple log file
3. Legacy scripts that expect `nohup` semantics

Examples with Explanations

Example: basic background job

```
nohup ./long-job.sh &
```

Starts the job, ignores hangup, backgrounds it. Output typically lands in `nohup.out` in the current directory.

Example: explicit log file

```
nohup ./long-job.sh > /var/tmp/long-job.log 2>&1 &  
echo $!
```

Always redirect in real use so logs do not scatter into surprise `nohup.out` files. `$!` is the background PID.

Example: find and stop later

```
nohup python3 train.py --epochs 100 > train.log 2>&1 &  
echo $! > train.pid  
  
# later:  
kill -TERM "$(cat train.pid)"  
# wait, then kill -KILL if needed
```

Record the PID; do not rely on grepping vaguely for `python3`.

Example: disown alternative (bash)

```
./long-job.sh > job.log 2>&1 &  
disown
```

`disown` removes the job from the shell's job table so logout does not send HUP to it. Different mechanism than `nohup`, similar operator goal.

Example: combine with nice / ionice

```
nohup nice -n 19 ionice -c3 ./batch-compress.sh > batch.log 2>&1 &
```

Low CPU and idle I/O class — polite neighbor on shared hosts.

Example: remote one-liner over SSH

```
ssh host 'nohup /opt/app/rebuild.sh > /tmp/rebuild.log 2>&1 &'
```

SSH can still race with job startup; for reliability prefer `tmux new -d` or a systemd unit.

Example: check nohup.out

```
nohup seq 1 1000000 &  
tail -f nohup.out
```

Demonstrates default output capture when no redirection is given.

Safety

- `nohup` does **not** restart on crash, reboot, or OOM kill — not a supervisor.
- Writing `nohup.out` into a shared or production cwd can fill disks or leak sensitive prints.
- Background jobs may still receive other signals; only HUP is specially handled by `nohup`.
- Root + `nohup` + destructive scripts is still destructive after you disconnect.

Notes & Pitfalls

- Prefer `tmux new -s name` for anything you might need to reattach to interactively.
- Prefer `systemd-run --user` or a real unit for recurring or critical jobs.
- Relative paths in the command resolve at start time; if you `cd` later in another shell, the job's cwd is unchanged.
- `nohup.out` is **append** mode — old runs accumulate; rotate or redirect explicitly.
- Some environments use `setsid` or `ssh -f` patterns instead; know which layer owns the process.

Related Commands

- `tmux` / `screen` — reattachable sessions
- `systemd-run` / `systemctl` — supervised execution
- `disown` — bash job-table detach
- `setsid` — new session
- `timeout` — bound runtime
- `nice` / `ionice` — resource politeness

Additional Resources

- `man nohup`
- `man systemd-run`

timeout

Overview

`timeout` runs a command and sends a signal if it still runs after a duration. Use it to bound hung network tools, flaky tests, or “should never take this long” maintenance scripts. Default kill signal is **TERM**; you can escalate to **KILL** after a grace period.

Syntax

```
timeout [options] DURATION COMMAND [ARG]...
```

Duration format

Form	Meaning
10	10 seconds
10s	10 seconds
5m	5 minutes
2h	2 hours
1d	1 day
0 / unset special	See man page — 0 disables the timeout on GNU coreutils

Floating point seconds are accepted (e.g. 0.5).

Common Options

Option	Description
<code>-s SIGNAL, --signal=SIGNAL</code>	Signal on timeout (default TERM)
<code>-k DURATION, --kill-after=DURATION</code>	After first signal, wait then send KILL
<code>-f, --foreground</code>	Do not run command in a separate group (foreground quirks)
<code>-p, --preserve-status</code>	Exit with the command’s status even on timeout

Option	Description
<code>-v, --verbose</code>	Diagnose signal sends
<code>--</code>	End of options (when command looks like a flag)

Exit status (GNU coreutils)

Status	Meaning
124	Command timed out (default)
125	<code>timeout</code> itself failed
126	Command found but not invocable
127	Command not found
137	Often command killed by SIGKILL (128+9) depending on path
other	Exit status of <code>COMMAND</code> if it finished in time

With `--preserve-status`, a timed-out command may return the status as if it received the signal (e.g. 128+signal) rather than 124 — check `man timeout` for your coreutils version.

Safety

- Bound commands that talk to **production** carefully — abrupt `TERM` can leave partial migrations; prefer app-level deadlines when available.
- Always consider `-k` so wedged processes that ignore `TERM` do not hang your automation forever.
- `timeout` is not a substitute for proper service health checks and systemd `TimeoutStopSec=`.

Key Use Cases

1. Cap flaky CI steps and network probes
2. Prevent hung `fsck`, `scp`, or package hooks from blocking unattended jobs
3. Force-fail long unit tests
4. Demo/debug with short lifetimes

Examples with Explanations

Example: basic time limit

```
timeout 10s ping -c 1000 1.1.1.1
```

Stops ping after 10 seconds even if count is not finished.

Example: TERM then KILL grace period

```
timeout -k 5s 30s ./migrate.sh
```

At 30s send **TERM**; if still alive 5s later, send **KILL**. Default operator pattern for stubborn processes.

Example: choose the first signal

```
timeout -s INT 20s ./app  
timeout --signal=TERM -k 10s 2m ./app
```

INT mimics Ctrl-C; TERM is the usual graceful stop.

Example: capture timeout in scripts

```
if timeout 1m ./backup.sh; then  
    echo "backup ok"  
else  
    rc=$?  
    if [ "$rc" -eq 124 ]; then  
        echo "backup timed out" >&2  
    else  
        echo "backup failed rc=$rc" >&2  
    fi  
    exit "$rc"  
fi
```

Treat 124 as a distinct failure mode in monitoring and retries.

Example: verbose diagnosis

```
timeout -v -k 2s 5s sleep 60
```

Shows which signals were sent when.

Example: network tool guardrails

```
timeout 15s curl -fsS https://example.com/health  
timeout 5s dig +time=2 +tries=1 example.com
```

Keeps cron probes from stacking up when DNS or HTTP hangs.

Example: command that starts with a dash

```
timeout 10s -- ./mytool --dangerous-flag
```

-- prevents option eating by timeout.

Example: foreground mode

```
timeout --foreground 30s vim file.txt
```

Sometimes needed for programs that expect to be session leaders or read the TTY specially; default separate process group is usually correct for daemons/scripts.

Notes & Pitfalls

- Children that **double-fork** or reparent may outlive the timed command group depending on how they detach — design jobs not to daemonize if you need hard bounds.
- Pipeline gotcha: `timeout 5s cmd1 | cmd2` only times out `cmd1` unless you structure differently (`timeout 5s bash -c 'cmd1 | cmd2'`).
- systemd units have their own start/stop timeouts; combine thoughtfully.
- BusyBox `timeout` option sets are smaller — stick to common flags in portable scripts.

Related Commands

- `kill` — manual signaling
- `timelimit` — alternative on some systems
- `systemd-run -p RuntimeMaxSec=` — cgroup-enforced limits
- `nice` / `ionice` — resource politeness without a deadline
- `nohup` — survive hangup (orthogonal concern)

Additional Resources

- `man timeout`

watch

Overview

`watch` runs a command repeatedly, showing fullscreen output so you can see changes over time. Ideal for ad-hoc monitoring (`df`, `kubectl get`, `systemctl status`) without writing a loop. Not a metrics system — for long-term graphs use Prometheus/node_exporter or similar.

Syntax

```
watch [options] COMMAND
watch [options] -- COMMAND
```

COMMAND is passed to `sh -c` by default, so shell features work inside the watched string.

Common Options

Option	Description
<code>-n SECS, --interval SECS</code>	Seconds between runs (default 2; fractions OK on modern procs)
<code>-d, --differences</code>	Highlight differences between runs
<code>-d=permanent</code>	Keep highlighting changes persistently
<code>-t, --no-title</code>	Hide header (interval, host, command)
<code>-b, --beep</code>	Beep if command exits non-zero
<code>-e, --errexit</code>	Freeze screen on non-zero exit
<code>-g, --chgexit</code>	Exit when output changes
<code>-c, --color</code>	Interpret ANSI color sequences
<code>-x, --exec</code>	Pass command to <code>exec</code> instead of <code>sh -c</code>
<code>-p, --precise</code>	Attempt to run every interval accounting for command runtime

Key Use Cases

1. Watch disk, memory, or service state during an incident
2. Wait until a condition changes (`-g`) then continue a scripted workflow

3. Highlight diffs while configs or replica counts converge
4. Lightweight “dashboard” over SSH without extra tools

Examples with Explanations

Example: basic interval monitor

```
watch -n 1 df -h
```

Refresh disk usage every second.

Example: highlight changes

```
watch -d free -h  
watch -d 'ip -s link show eth0'
```

Changed characters flash between iterations — great for counters and free memory.

Example: multi-command pipeline

```
watch -n 2 'ps aux --sort=-%cpu | head -n 15'
```

Quote the whole pipeline so the shell inside `watch` owns the pipe.

Example: systemd / journal peek

```
watch -n 2 'systemctl --failed --no-pager'  
watch -n 1 'systemctl is-active nginx; systemctl status nginx --no-pager | head -n 15'
```

Handy while restarting services.

Example: exit when output changes

```
watch -g -n 1 'cat /var/run/myapp/ready'  
echo "ready flag changed"
```

Blocks until the command’s output differs from the first run, then exits. Useful as a crude barrier in shell workflows.

Example: stop on error

```
watch -e -n 5 './healthcheck.sh'
```

Freezes on first non-zero exit so you can read the failure output.

Example: colors from modern CLIs

```
watch -c -n 2 'ls --color=always -l /srv/data'
```

Without `-c`, ANSI escapes may show as garbage.

Example: precise interval

```
watch -p -n 5 'date +%T; ./poll.sh'
```

Tries to keep wall-clock cadence even when `poll.sh` takes a second or two (not a hard realtime guarantee).

Example: exec mode (no shell)

```
watch -x -n 1 ping -c 1 1.1.1.1
```

Avoids `sh -c` — safer when you do not want shell expansion, but pipelines/globs will not work.

Notes & Pitfalls

- Default shell is `sh -c`; quote carefully and avoid unescaped `$` that expand **before** `watch` runs (use single quotes for the outer command string).
- Long-running watched commands that exceed the interval will stack poorly without `-p` discipline; keep the inner command fast.
- `watch` clears the screen — redirect-friendly logging needs a different approach (`while sleep; do ...; done >> log`).
- Exit with `Ctrl-C`.
- On minimal systems ensure `procps/procps-ng` is installed (Ubuntu has `watch` from that package).

Related Commands

- `while sleep N; do ...; done` — scriptable loop with logging
- `vmstat 1 / iostat -xz 1` — specialized continuous metrics
- `journalctl -f / tail -f` — follow streams
- `systemd-run / timers` — scheduled work
- `timeout` — bound a single run

Additional Resources

- `man watch`

sleep

Overview

`sleep` suspends execution for a specified time. Used in shell scripts for pacing, simple retries, and demos. For precise timers, event-driven waits, or jittered backoff, higher-level tools may be better — but `sleep` remains the universal delay primitive.

Syntax

```
sleep NUMBER[SUFFIX]...
```

GNU `sleep` accepts floating point and multiple arguments (sums them).

Time suffixes (GNU)

Suffix	Unit
s	Seconds (default)
m	Minutes
h	Hours
d	Days

Examples with Explanations

Basics

```
sleep 5
sleep 0.5
sleep 2m
sleep 1h
sleep 1m 30s           # GNU: 90 seconds total
```

Simple retry loop

```
until ping -c1 -W1 gateway >/dev/null; do
    sleep 2
done
```

Exponential backoff (bash)

```
delay=1
for i in 1 2 3 4 5; do
    curl -fsS https://example.com/ && break
    sleep "$delay"
    delay=$((delay * 2))
done
```

Rate limiting

```
while read -r host; do
    ssh "$host" 'uname -r'
    sleep 0.2
done < hosts.txt
```

Background heartbeat

```
while true; do
    date -Is >> /tmp/heartbeat
    sleep 60
done &
```

Prefer systemd timers for production heartbeats.

Interruptible

```
sleep 300
# Ctrl-C sends SIGINT to foreground sleep
```

Portable note

```
# POSIX: integer seconds only on some platforms
sleep 5
# fractional may require GNU coreutils
```

Notes / Pitfalls

- Busy loops without sleep burn CPU — always throttle polls.
- `sleep infinity` works on GNU; not portable — use large numbers or `tail -f /dev/null`.
- Sleep drift: not a real-time scheduler; long loops accumulate error.
- Don't use sleep as the only readiness check for services — probe health endpoints.
- In pipelines, know which process sleeps.

2026-relevant notes

- Prefer `systemd-run` timers / `cron` / `watch` for periodic work.
- Kubernetes readiness/liveness already handle waits — avoid crude sleep in hot paths.
- `timeout` pairs with commands that must not hang forever.

Related Commands

- `timeout` — bound a command's runtime
- `watch` — run a command repeatedly
- `usleep` — legacy microseconds (often gone; use `sleep 0.001`)
- `at` / `systemd-run` — schedule future work
- `wait` — wait for jobs, not wall clock alone

Additional Resources

- `man sleep`

nice

Overview

`nice` starts a command with a modified **CPU niceness** — a soft priority bias for the scheduler. Higher nice means **more polite** (less CPU when others compete); lower (negative) nice means more aggressive. Range is typically **-20..19**, default **0**. Unprivileged users can only **increase** niceness. Use `renice` to change a running process; use `ionice` for disk I/O class.

Syntax

```
nice [OPTION] [COMMAND [ARG]...]
nice -n ADJUSTMENT COMMAND [ARG]...
```

With no command, some implementations print the current niceness.

Related:

```
renice [-n] PRIORITY -p PID...
renice [-n] PRIORITY -u USER...
```

Common Options

Option	Description
<code>-n ADJ</code> , <code>--adjustment=ADJ</code>	Add ADJ to default nice (often written as absolute intent: <code>nice -n 10</code>)
<code>--help</code> / <code>--version</code>	Meta

`renice` selection:

Option	Description
<code>-p PID</code>	Process ID(s)
<code>-g PGID</code>	Process group
<code>-u USER</code>	All of user's processes
<code>-n PRI0</code>	Absolute nice value to set (util-linux style)

Safety

- Negative nice values need **root** (or **CAP_SYS_NICE**) — do not renice critical daemons downward without understanding latency impact.
- Nice is **not** a hard cap or cgroup limit; a niced process can still use 100% CPU when the machine is idle.
- For multi-tenant fairness prefer **cgroups / systemd resource controls** (**CPUWeight=**, **CPUQuota=**) over global renice.

Key Use Cases

1. Run backups/compiles politely during business hours
2. Deprioritize a runaway batch without killing it
3. Slightly boost an interactive build only when policy allows
4. Inspect niceness during incident triage

Examples with Explanations

Example: start a polite batch job

```
nice -n 19 ./batch-compress.sh
nice -n 10 tar -czf /backup/home.tgz /home
```

19 is maximum politeness on Linux. Good default for “finish eventually, don’t fight production”.

Example: background + log

```
nice -n 15 ./reindex.sh > /var/tmp/reindex.log 2>&1 &
```

Combine with shell backgrounding; still not a supervisor.

Example: renice a running PID

```
ps -o pid,ni,cmd -p 1234
renice -n 15 -p 1234
sudo renice -n -5 -p 1234
```

Ordinary users can raise nice (e.g. 0 → 15) on their own processes; lowering requires privilege.

Example: renice all of a user's jobs

```
sudo renice -n 10 -u builduser
```

Useful when a CI account is starving interactive sessions.

Example: observe nice in ps / top

```
ps -eo pid,user,ni,pri,cmd --sort=ni | head
ps -o pid,ni,comm -p 1234
# top/htop: NI column; htop F7/F8 adjust interactively
```

NI is niceness; PRI is the kernel priority derived from scheduling policy and nice.

Example: nice + ionice + nohup combo

```
nohup nice -n 19 ionice -c3 ./nightly-rsync.sh > /var/log/nightly-rsync.log 2>&1 &
```

CPU polite + idle I/O class + survive hangup — classic overnight job pattern.

Example: systemd equivalent knobs

```
systemd-run --user --nice=15 ./batch.sh
# in a unit:
# Nice=15
# CPUWeight=50
```

Prefer unit settings for services you own long-term.

Example: print current nice (when supported)

```
nice
```

Some systems print the shell's current niceness if no command is given.

Understanding Output

- `nice` itself is silent on success when launching a command.
- Failed `renice` prints permission or `ESRCH` errors.
- Inherited niceness: children typically inherit the parent's nice value.

Notes & Pitfalls

- Syntax confusion: old forms like `nice -10 cmd` mean adjustment **-10** on some Unixes; on GNU prefer `nice -n 10 cmd` explicitly.
- Realtime policies (`SCHED_FIFO` via `chrt`) override ordinary nice semantics — different tool.
- Containers may restrict `CAP_SYS_NICE`; negative `renice` fails even as “root” in a locked-down pod.
- Autogroup / CFS details can make nice effects less dramatic than expected on desktop kernels — still useful under contention.

Related Commands

- `renice` — change running processes
- `ionice` — I/O scheduling class
- `chrt` — realtime/other scheduling policies
- `timeout` — bound wall time
- `systemctl` / `cgroups` — proper resource control
- `top` / `htop` / `ps` — observe NI

Additional Resources

- `man nice`
- `man renice`
- `man sched(7)`

renice

Overview

The **renice** command alters the scheduling priority of running processes. It allows you to change the nice value of processes that are already running.

Syntax

```
renice [-n] priority [-g|-p|-u] identifier...
```

Common Options

Option	Description
-n	Specify nice increment
-g	Interpret as process groups
-p	Interpret as process IDs
-u	Interpret as usernames
--help	Display help
--version	Show version

Nice Values

- Range: -20 to 19
- Lower values = higher priority
- Higher values = lower priority
- Only root can set negative values
- Default: current nice value

Key Use Cases

1. Adjust process priority
2. Resource management
3. Performance tuning
4. System optimization
5. Process control

Examples with Explanations

Example 1: Basic Usage

```
renice +5 -p 1234
```

Change priority of PID 1234

Example 2: User Processes

```
renice 10 -u username
```

Change priority of user's processes

Example 3: Process Group

```
renice -5 -g 100
```

Change priority of process group

Understanding Output

Format:

```
1234: old priority 0, new priority 5
```

Error messages for: - Permission denied - Invalid priority - Process not found - User not found

Common Usage Patterns

1. Lower process priority:

```
renice +10 -p $(pgrep firefox)
```

2. Increase user priority:

```
sudo renice -5 -u apache
```

3. Multiple processes:

```
renice 5 -p 1234 5678
```

Performance Analysis

- Priority impact
- System load effect
- Resource allocation
- Process behavior
- Scheduling changes

Related Commands

- `nice` - Start with priority
- `top` - Process monitoring
- `ps` - Process status
- `ionice` - I/O scheduling
- `chrt` - Real-time scheduling

Additional Resources

- [Renice Manual](#)
- [Process Priority Guide](#)
- [CPU Scheduling](#)

Best Practices

1. Monitor system impact
2. Document changes
3. Consider dependencies
4. Regular review
5. Use appropriate values

Use Cases

1. Performance tuning
2. Resource allocation
3. System maintenance
4. Service optimization
5. Troubleshooting

ionice

Overview

`ionice` sets or queries the **I/O scheduling class and priority** of a process (Linux CFQ/BFQ-oriented interface; behavior depends on the current I/O scheduler and kernel). Use it so bulk disk work (backups, `updatedb`, large `rsync`) yields to latency-sensitive services. Orthogonal to CPU `nice`.

Syntax

```
ionice [options] -p PID...
ionice [options] COMMAND [ARG]...
ionice -p PID
```

I/O classes

Class	-c	Meaning
None	0	No class set / request default handling
Realtime	1	Highest access (requires privilege; can starve others)
Best-effort	2	Default class; priority 0–7 within class
Idle	3	Only when disk is idle — safest for bulk jobs

Best-effort priority: **0** = highest within class, **7** = lowest (opposite intuition from CPU `nice` numbers — read twice).

Common Options

Option	Description
<code>-c CLASS</code>	Class number or name (<code>idle</code> , <code>best-effort</code> , <code>realtime</code>) on newer <code>util-linux</code>

Option	Description
<code>-n</code> <code>PRIORITY</code>	Priority 0–7 within best-effort or realtime
<code>-p</code> <code>PID</code>	Act on existing PID (repeatable)
<code>-t</code>	Ignore failures (e.g. unsupported) — “try”
<code>-P</code> <code>PGID</code>	Process group (where supported)
<code>-u</code> <code>UID</code>	As user (where supported)

Safety

- **Realtime I/O (`-c1`)** can starve the system — avoid on production unless you know the storage stack and blast radius.
- Prefer **idle (`-c3`)** for overnight compaction, virus scans, and catalog updates.
- Capability/`CAP_SYS_ADMIN` or root may be required for realtime class; idle/best-effort for own processes is usually fine.
- On modern multi-queue SSDs with certain schedulers, ionice effects can be **weaker** than on classic CFQ HDDs — still good hygiene for HDDs and shared spindles.

Key Use Cases

1. Start backups in the idle I/O class
2. Soften a live `rsync/tar` hurting databases
3. Query current I/O class of a suspect PID
4. Combine with `nice` for “be a good neighbor” batch windows

Examples with Explanations

Example: run command as idle I/O

```
ionice -c3 ./backup.sh
ionice -c idle ./backup.sh
```

Only issues disk I/O when the device is not busy with others — best default for bulk work.

Example: best-effort low priority

```
ionice -c2 -n7 tar -czf /backup/data.tgz /data
```

Class best-effort, lowest priority level. Less extreme than idle; still polite.

Example: change a running process

```
ionice -c3 -p 1234
ionice -c2 -n7 -p 1234
```

Apply without restarting. Verify with a query afterward.

Example: query current setting

```
ionice -p 1234
# example output: none: prio 0
# or: idle
# or: best-effort: prio 4
```

Silent context for incident notes: “PID 1234 was idle class during the spike”.

Example: nice + ionice together

```
nice -n 19 ionice -c3 rsync -a /data/ /mnt/backup/data/
```

CPU and disk both deprioritized. Standard pattern for shared hosts.

Example: nohup overnight job

```
nohup nice -n 19 ionice -c3 \
  rsync -a --delete /srv/ /mnt/cold/srv/ \
  > /var/log/rsync-cold.log 2>&1 &
```

Survives logout, polite CPU/I/O, logged.

Example: try without failing scripts

```
ionice -t -c3 ./portable-job.sh
```

-t ignores failures when ionice cannot set class (containers, odd kernels) so the job still runs.

Example: avoid realtime unless justified

```
# dangerous on shared storage - shown for completeness only
# sudo ionice -c1 -n0 -p "$LATENCY_CRITICAL_PID"
```

Prefer storage QoS, cgroup `io.latency/IOWeight=`, and proper service isolation instead.

Notes & Pitfalls

- **Not portable** to non-Linux; no-op or absent on macOS/BSD.
- Effect depends on **scheduler** (`mq-deadline`, `bfq`, `none` for NVMe, etc.). Check `/sys/block/*/queue/scheduler`.
- Children inherit I/O priority at fork time; later `ionice -p` on the parent does not always retarget already-spawned children — apply to the whole tree if needed.
- systemd: `IOSchedulingClass=` / `IOSchedulingPriority=` in units; often cleaner than wrappers.
- Do not confuse with `nice` (CPU) or `chrt` (CPU policy).

Related Commands

- `nice` / `renice` — CPU niceness
- `chrt` — CPU scheduling policy
- `ionice` + `nohup` / `tmux` — long batch sessions
- `iostat` / `iotop` — observe disk pressure
- `systemctl` — unit-level I/O settings

Additional Resources

- `man ionice`
- `man ioprio_set` (syscall background)

fuser

Overview

fuser lists PIDs using a file, mount point, or network port. It is a concise complement to **lsof**: faster for “who is blocking my umount?” and optional signaling of those processes.

Syntax

```
fuser [options] name...
```

Common Options

Option	Description
-v	Verbose (USER, PID, ACCESS, COMMAND)
-m	Argument is a mount / filesystem
-k	Kill processes (default SIGKILL with -k alone on many builds— confirm man)
-i	Interactive confirm before kill
-n tcp\ udp	Namespace for ports
-u	Append user
-4 / -6	Address family
-M	Check is a mountpoint
signal	e.g. -TERM before -k on some implementations

Access codes (verbose): **c** current dir, **e** executable, **f** open file, **r** root dir, **m** mmap, etc.

Key Use Cases

1. Find processes preventing **umount**
2. See who holds a TCP/UDP port
3. Signal processes holding a lock file

Examples with Explanations

Mount busy

```
sudo fuser -vm /mnt/data
sudo fuser -vm /mnt/data # review
# last resort:
sudo fuser -kim /mnt/data # interactive kill on mount
```

Ports

```
sudo fuser -v 22/tcp
sudo fuser -v 8080/tcp
sudo fuser -n tcp 5432
```

Prefer also: `ss -ltnp | grep 8080`.

Files

```
fuser -v /var/log/app.log
fuser -ki file.lock
```

One-liner recipes

```
# Before umount
sudo fuser -vm /mnt && sudo umount /mnt

# Who owns this listening port?
sudo ss -ltnp | grep ':8080'
sudo fuser -v 8080/tcp
```

Notes & Pitfalls

- **-k is destructive**—always `-v` first; prefer `-i`.
- Default signal behavior differs slightly by util-linux version—read `man fuser` on the host.
- Network namespace: ports need `tcp/udp` suffix or `-n`.
- Containers: `fuser` inside vs on host sees different process IDs/namespaces.

2026-relevant notes

- For listeners, **ss -lntp** is usually clearer than **fuser** alone.
- Combine with **lsof -t** for scripts that need PID lists.
- Lazy **umount** (**umount -l**) hides problems—fix the holders when possible.

Related Commands

- **lsof** — detailed open files
- **ss** — sockets
- **umount** — detach mounts
- **fuser -k** vs **kill** — scope differs

Additional Resources

- `man fuser`

ulimit

Overview

`ulimit` is a **shell builtin** (bash/zsh/etc.) that shows or sets **resource limits** for the current shell and its child processes: open files, processes, stack size, core dumps, and more. It does not permanently reconfigure the whole system—use `/etc/security/limits.conf`, systemd unit directives, or cgroups for persistence and service-wide policy.

Syntax

```
ulimit [-SHacdefilmnpqrstuvx] [limit]
```

Common Options

Option	Resource
-a	Show all limits
-n	Max open file descriptors
-u	Max user processes
-v	Address space (KB) where supported
-f	Max file size (blocks)
-c	Core file size
-s	Stack size
-l	Locked memory
-m	Resident set (often ignored on Linux)
-t	CPU seconds
-H	Hard limit
-S	Soft limit

Soft limits can be raised up to the hard limit by the user; hard limits need root (or were set that way at login).

Key Use Cases

1. Raise `nofile` for databases, proxies, load generators
2. Allow core dumps (`-c unlimited`) while debugging
3. Diagnose “Too many open files” / “cannot fork”
4. Compare shell limits vs systemd service limits

Examples with Explanations

Show limits

```
ulimit -a
ulimit -n
ulimit -u
ulimit -H -n
ulimit -S -n
```

Raise open files (session)

```
ulimit -n 65536
ulimit -n
```

Fails if hard limit is lower—check `ulimit -H -n` and raise via `limits.d` or `systemd`.

Cores and stack

```
ulimit -c unlimited
ulimit -s 8192
```

One-liner recipes

```
# Reproduce app under higher nofile
ulimit -n 100000 && ./my-server

# Show soft/hard nofile pair
printf 'soft=%s hard=%s\n' "$(ulimit -Sn)" "$(ulimit -Hn)"

# Compare with PID 1 or a service (systemd)
systemctl show nginx -p LimitNOFILE --value
cat /proc/$(pgrep -n nginx)/limits
```

Persistent configuration (context)

```
# /etc/security/limits.d/99-nofile.conf (pam_limits - login sessions)
# myuser soft nofile 65536
# myuser hard nofile 1048576

# systemd service drop-in
# [Service]
# LimitNOFILE=65536
```

`ulimit` alone in `~/.bashrc` does **not** affect `systemd` services.

Notes & Pitfalls

- **Builtin:** not `/usr/bin/ulimit`; `dash`/`busybox` shells differ.
- **Non-interactive SSH** may skip `bashrc`—put service limits in `systemd/unit` files.
- Containers inherit `cgroup` and `docker --ulimit` settings; shell `ulimit` cannot exceed them.
- `unlimited` is not always valid for every resource.
- Changing `-n` in one terminal doesn't affect already-running daemons.

2026-relevant notes

- Prefer `systemd LimitNOFILE=`, `TasksMax=`, and `cgroup v2` pressure metrics over only `pam` limits on pure server images.
- Kubernetes sets pod resource limits separately; `ulimit` inside a pod is still constrained by the container runtime.
- Kernel `fs.file-max` and per-user limits both matter for “too many open files”.

Related Commands

- `prlimit` — get/set limits of a running PID
- `systemctl show` — unit `Limit*` properties
- `cat /proc/PID/limits` — effective process limits
- `sysctl fs.file-max` — system-wide file ceiling

Additional Resources

- `help ulimit (bash)`
- `man limits.conf`, `man systemd.exec`

ipcs

Overview

`ipcs` lists **System V IPC** objects: message queues, shared memory segments, and semaphore arrays. Useful when debugging legacy applications, databases, or middleware that use SysV IPC, and when cleaning up leftover segments after crashes.

POSIX shared memory and mutexes use different interfaces (`/dev/shm`, `pthread`s) — not shown by `ipcs`.

Syntax

```
ipcs [options]
```

Common Options

Option	Description
<code>-a</code>	All resources (default summary style)
<code>-q</code>	Message queues
<code>-m</code>	Shared memory
<code>-s</code>	Semaphores
<code>-t</code>	Time info
<code>-p</code>	PIDs (creator/last operator)
<code>-c</code>	Creator user/group
<code>-l</code>	Resource limits
<code>-u</code>	Summary usage
<code>-i id</code>	Details for specific ID
<code>-b</code>	Brief bytes columns where applicable

Examples with Explanations

Overview

```
ipcs
ipcs -a
ipcs -m
ipcs -q
ipcs -s
```

Limits and usage

```
ipcs -l
ipcs -u
```

Creator and PIDs

```
ipcs -m -p
ipcs -m -c
ipcs -i 123456 -m
```

Find leftovers by user

```
ipcs -m | awk 'NR>3 && $3=="alice" {print}'
```

Cleanup companion

```
ipcs -m
sudo ipcrm -m <shmid>
# or by key:
sudo ipcrm -M 0x00001234
```

See `ipcrm` page for deletion safety.

Kernel params (context)

```
sysctl kernel.shmmax kernel.shmall kernel.shmmni
sysctl kernel.msgmax kernel.msgmni
sysctl kernel.sem
```

Notes / Pitfalls

- Deleting live shared memory of a running DB will wreck it — identify owners first.
- IDs vs keys: `ipcrm` flags differ (`-m id` vs `-M key`).
- Docker/IPC namespaces hide or isolate objects — empty `ipcs` may be namespace-related.
- Modern apps often prefer POSIX shm (`/dev/shm`) and won't appear here.
- Permissions: you may only see objects you can access.

2026-relevant notes

- Still relevant for Oracle, older SAP, and legacy C apps; less so for greenfield Go/Node services.
- Prefer container IPC isolation over host-global SysV segments when designing new systems.
- Monitor with app-native tools first; use `ipcs` for orphan cleanup after hard crashes.

Related Commands

- `ipcrm` — remove IPC objects
- `ls -l /dev/shm` — POSIX shared memory files
- `sysctl` — kernel IPC limits
- `fuser` / `lsof` — who uses files (not always SysV)
- `pmap` — process memory maps

Additional Resources

- `man ipcs`, `man ipc`

ipcrm

Overview

ipcrm removes **System V IPC** objects: shared memory segments, semaphore arrays, and message queues. Use after confirming with **ipcs** that objects are orphaned — never delete resources still owned by live production processes.

Syntax

```
ipcrm [options]
ipcrm -m id | -M key
ipcrm -q id | -Q key
ipcrm -s id | -S key
```

Common Options

Option	Description
-m shmid	Remove shared memory by ID
-M key	Remove shared memory by key
-q msqid	Remove message queue by ID
-Q key	Remove message queue by key
-s semid	Remove semaphore by ID
-S key	Remove semaphore by key
-a	Remove all (dangerous; some versions / with filters)
-v	Verbose

Exact **-a** behavior varies — read `man ipcrm` before bulk deletes.

Examples with Explanations

Inspect then remove one segment

```
ipcs -m
sudo ipcrm -m 98305
ipcs -m
```

Remove by key

```
ipcs -m
# note the key column (hex)
sudo ipcrm -M 0x00000000
```

Message queues / semaphores

```
ipcs -q
sudo ipcrm -q 12
ipcs -s
sudo ipcrm -s 34
```

Scripted cleanup of your own leftovers

```
# example: remove shared memory owned by current user carefully
id -u
ipcs -m -c
# manually select ids, then:
# ipcrm -m ID
```

Automating bulk ipcrm without filters is risky on multi-app hosts.

After crash lab example

```
# app crashed without IPC_RMID
ipcs -m -p
# verify no live PID uses it
ps -p <cpid>,<lpid>
sudo ipcrm -m <shmids>
```

Notes / Pitfalls

- Wrong ID → impact on another application (databases!).
- Need permission; root can remove almost anything.
- Race with process still attaching — prefer graceful app shutdown.
- Keys of 0x00000000 may mean `IPC_PRIVATE` — identify carefully.
- Does not remove POSIX shm files in `/dev/shm` — use `rm` there if appropriate.

2026-relevant notes

- Prefer designs that set `IPC_RMID` or use POSIX shm with clear lifecycle.
- In containers, IPC namespace teardown often cleans objects automatically on exit.
- Document any required SysV objects for legacy apps so ops doesn't “clean” them.

Related Commands

- `ipcs` — list IPC
- `sysctl` — kernel limits
- `ls /dev/shm` — POSIX shm
- `rm` — remove POSIX shm files
- `ipcmk` — create IPC (rarely needed manually)

Additional Resources

- `man ipcrm`

System Monitoring

Intro

Intro

Measure subsystem health over time and debug *why* a host is slow or stuck: virtual memory, disks, CPUs, open files, and syscall/library traces.

Commands in this part

Command	Role
vmstat	vmstat reports virtual memory, process run/block queues, CPU, and I/O counters in a compact table.
iostat	iostat (from the sysstat package) reports CPU utilization and per-device I/O rates, queue depth, await, and utilization.
mpstat	mpstat (sysstat) reports CPU utilization, optionally per logical CPU, so you can see single-thread bottlenecks,...
sar	sar (System Activity Reporter, sysstat) prints live and historical metrics: CPU, memory, load, disk, network, and more.
lsof	lsof lists open files.
strace	strace traces system calls and signals for a process.
perf	perf is Linux's primary performance analysis suite.
btopy	btopy is a modern, interactive resource monitor with a polished TUI: CPU (often per-core with clocks/temps when...
ltrace	ltrace intercepts and prints dynamic library calls of a process (similar to how strace prints syscalls).

Suggested starting points

1. Load vs I/O vs CPU: `vmstat`, `iostat`, `mpstat`, `sar`.
2. Who holds a file or port: `lsof`.
3. Why a process fails: `strace` (syscalls), `ltrace` (library calls).
4. Hotspots: `perf` (and interactive `bt` for a dashboard).

Related parts

- Processes and jobs — PIDs and signals
- Storage and filesystems — disk health and space
- Networking — `ss`/`tcpdump` for network-bound issues

Continue with the individual command pages in this part.

vmstat

Overview

`vmstat` reports virtual memory, process run/block queues, CPU, and I/O counters in a compact table. Ideal for a quick terminal loop that answers: **CPU-bound, RAM pressure, or disk wait?** Prefer sample mode (`delay count`) over a single line — the first line is often averages since boot.

Syntax

```
vmstat [options] [delay [count]]
```

Common Options

Option	Description
<code>delay count</code>	Sample every <i>delay</i> seconds, <i>count</i> times
<code>-w</code>	Wide columns (avoid truncated fields)
<code>-S k\ K\ m\ M</code>	Unit scaling for memory fields
<code>-a</code>	Active/inactive memory instead of buff/cache split detail
<code>-s</code>	One-shot event counter summary
<code>-d</code>	Disk statistics
<code>-D</code>	Disk summary totals
<code>-t</code>	Timestamp each sample line
<code>-n</code>	Header once (not every screenful)
<code>-f</code>	Fork count since boot

Install note: `vmstat` is from **procps** (preinstalled on Ubuntu). Disk/historical depth may still push you to `sysstat` (`iostat`, `sar`).

Key Use Cases

1. Live “why is it slow?” triage loop
2. Detect swap thrash (`si/so`)

3. Spot I/O wait vs CPU user/system
4. VM steal time (st) on cloud guests

Examples with Explanations

Live sample (default triage)

```
vmstat 1 10
vmstat -w 1 5
```

One-second samples, ten lines (or five wide). **Ignore the first data line** when using a delay — it is often since-boot averages; subsequent lines are interval deltas.

Timestamped capture for logs

```
vmstat -t 2 30 | tee /tmp/vmstat.out
```

Useful during an incident window you will review later.

Memory summary counters

```
vmstat -s
vmstat -s -S M
```

Cumulative counters (forks, pages paged in/out, etc.) — good for a snapshot, not a live rate loop.

Disk stats mode

```
vmstat -d 1 5
```

Per-disk reads/writes. For modern extended fields (await, %util), prefer `iostat -xz`.

Active / inactive memory view

```
vmstat -a 1 5
```

Highlights reclaimable cache vs active pages when diagnosing memory pressure.

Correlate with siblings

```
vmstat 1 5
iostat -xz 1 5
mpstat -P ALL 1 5
free -h
```

vmstat for system pulse; iostat for which disk; mpstat for per-CPU imbalance; free for human memory totals.

Understanding Output

Classic columns (names vary slightly by version/width):

Area	Fields	Operator hints
Procs	r , b	r > CPU count → runnable contention; b blocked (often I/O)
Memory	swpd , free , buff , cache	Rising swpd + low free → pressure (also check free -h)
Swap	si , so	Sustained non-zero swap in/out → RAM shortage / thrash
IO	bi , bo	Blocks in/out (not full disk latency picture)
System	in , cs	Interrupts, context switches (high cs can mean chatty scheduling)
CPU	us , sy , id , wa , st	% of time: user, kernel, idle, I/O wait , steal (hypervisor)

Pattern cheat sheet:

Pattern	Likely story	Next tool
High r , high us/sy , low id	CPU bound	mpstat -P ALL , pidstat/top
High b or wa	I/O bound	iostat -xz , iotop
Continuous si/so	Memory pressure	free -h , ps RSS, reduce cache pressure
High st	Hypervisor contention / undersized VM	host metrics, resize instance
High cs with low work	Possible scheduling noise	app profiling, perf

Units: without **-S**, memory fields are often KiB pages-related scaling depending on build — use **-w** and **-S M** when comparing to **free -h**.

Notes & Pitfalls

- **First line with delay** is historically since-boot — always take interval lines for “now”.
- **wa** means CPUs idle waiting on I/O, not “disk % busy”; a single slow device can elevate **wa**.
- **free** memory near zero can still be healthy if cache is reclaimable — watch **si/so** and application latency.
- Containers/cgroups: host **vmstat** sees the host; inside a container you still often see host-wide stats unless using cgroup-aware tools.
- Very short intervals (sub-second) add noise; 1s is a good default.

Related Commands

- **iostat** — per-device I/O rates and %util
- **mpstat** — per-CPU breakdown
- **free -h** — memory snapshot
- **top / htop / pidstat** — per-process
- **sar** — historical sysstat data
- **uptime** — load averages only

Additional Resources

- **man vmstat**
- **man free**

iostat

Overview

`iostat` (from the `sysstat` package) reports CPU utilization and **per-device** I/O rates, queue depth, await, and utilization. Use it to find busy disks, confirm I/O-wait hypotheses from `vmstat/top`, and compare devices under load. Install on Ubuntu with `sudo apt install sysstat`.

Syntax

```
iostat [options] [interval [count]]
iostat [options] [interval [count]] [device...]
```

Common Options

Option	Description
-x	Extended device stats (%util, await, aqu-sz, ...)
-z	Omit idle devices
-y	Omit the first since-boot report when using intervals
-d / -c	Devices only / CPU only
-h	Human-readable where supported
-k / -m	KB/s or MB/s units
-p	Include partitions
-t	Timestamps
-N	Map device-mapper names to prettier LV names
-j ID UUID ...	Display persistent device identifiers (version-dependent)

Key Use Cases

1. Live disk saturation checks (%util, await)
2. Confirm which device backs high wa in vmstat/top
3. Partition-level I/O (-p) during noisy-neighbor hunts
4. Scripted samples during load tests

Examples with Explanations

Extended live samples (default incident command)

```
iostat -xz 1 5  
iostat -xyz 1 10
```

-x extended fields, -z hide idle noise, interval 1s. Prefer -y (or ignore first report) so you are not reading since-boot averages.

Skip since-boot line explicitly

```
iostat -xyz 1 5  
# first sample after -y is the first real interval
```

One device (NVMe / disk)

```
iostat -xz 1 5 nvme0n1  
iostat -xz 1 5 sda
```

Focus when you already know the hot device from `lsblk`.

CPU only vs devices only

```
iostat -c 1 5  
iostat -d -xz 1 5
```

Partitions and mapper names

```
iostat -xzp 1 3  
iostat -xzN 1 3
```

-p splits partitions; -N helps on LVM hosts.

Timestamped capture

```
iostat -txz 1 30 | tee /tmp/iostat.out
```

Pair with process-level I/O

```
iostat -xz 1 5
# if installed:
sudo iotop -oPa
pidstat -d 1 5
```

iostat shows **devices**; iotop/pidstat show **who** is writing.

Understanding Output

CPU section resembles other sysstat tools (%user, %system, %iowait, %idle, %steal).

Extended device fields (names can vary slightly by sysstat version):

Field	Meaning
r/s w/s	Read/write requests completed per second
rkB/s wkB/s (or rMB/s)	Throughput
rrqm/s wrqm/s	Merged requests (elevator/scheduler merging)
await	Average time (ms) for requests (queue + service)
aqu-sz	Average queue length
rareq-sz wareq-sz	Average request size
%util	Share of time the device had outstanding I/O (~saturation signal for simple devices)

How to read it under fire:

Signal	Interpretation
%util ~ 100% and high await	Device saturated or slow
High await, modest %util	Latency issue (storage tier, queueing elsewhere, NFS)
High w/s + rising aqu-sz	Write burst / flush pressure
Only one partition hot	Noisy tenant or log volume on that part
%iowait high on CPU block	CPUs waiting on I/O — align with device rows

Caveats: on RAID, multipath, and NVMe with deep queues, %util is not a perfect “percent busy” for modern parallel devices — still useful, but combine with await and application latency.

Notes & Pitfalls

- Package: `sudo apt install sysstat` if command missing.
- Without interval, or the first line of interval mode, stats are often **since boot** — use intervals + -y.
- Device names from `lsblk`; cloud reattach can rename `sdX`.

- NFS latency does not show as a local block device — check client `mount` stats / server side.
- Containers: host-level `iostat` unless you instrument the host.

Related Commands

- `vmstat` — system-level `bi/bo` and `wa`
- `iotop` — per-process I/O (if installed)
- `pidstat -d` — per-process disk stats (`sysstat`)
- `lsblk` — map names to mounts
- `sar -d` — historical disk from `sysstat` logs
- `nfsiostat` — NFS-specific (`nfs-common`)

Additional Resources

- `man iostat`
- `man pidstat`

mpstat

Overview

`mpstat` (`sysstat`) reports **CPU utilization**, optionally **per logical CPU**, so you can see single-thread bottlenecks, softirq load, and steal time on individual cores. Aggregate tools (`vmstat`, `uptime`) hide imbalance; `mpstat -P ALL` reveals it. Install: `sudo apt install sysstat`.

Syntax

```
mpstat [options] [interval [count]]
```

Common Options

Option	Description
<code>-P ALL</code>	Each CPU plus all average row
<code>-P 0,2</code>	Specific CPU indices
<code>-u</code>	Utilization report (default)
<code>-I SUM\ CPU\ SCPU\ ALL</code>	Interrupt statistics
<code>-A</code>	Wide set of reports
<code>-o JSON</code>	JSON output (newer <code>sysstat</code>)
<code>-n</code>	Numeric only where applicable
<code>-T</code>	Topology-aware presentation (version-dependent)

Key Use Cases

1. Detect single-core saturation (one CPU 100%, others idle)
2. Softirq / network stack pressure (`%soft`)
3. Hypervisor steal (`%steal` / `%st`) per CPU
4. Interrupt distribution imbalances

Examples with Explanations

Average CPU samples

```
mpstat 1 5
```

Whole-system average every second — similar role to the CPU part of `vmstat`, with clearer labels.

Per-CPU breakdown (primary triage)

```
mpstat -P ALL 1 5  
mpstat -P ALL -u 1 3
```

Look for one or few CPUs pinned at high `%usr` while others idle → single-threaded or poorly parallelized work.

Specific cores

```
mpstat -P 0,1 1 5
```

Useful when pinning (taskset/cgroups) or NUMA-aware debugging.

Interrupts

```
mpstat -I SUM 1 3  
mpstat -I CPU 1 3  
mpstat -I SCPU 1 3
```

SUM totals; CPU / SCPU help when hunting IRQ imbalance or softirq storms (pair with `grep` in `/proc/interrupts`).

JSON for tooling

```
mpstat -P ALL -o JSON 1 1 | jq .
```

Pair with process CPU

```
mpstat -P ALL 1 5
pidstat -u 1 5
# top, then press 1 for per-CPU
```

mpstat shows **where** CPUs burn; pidstat/top show **who**.

Understanding Output

Common utilization fields:

Field	Meaning
%usr	User-space applications
%nice	User-space nice'd
%sys	Kernel
%iowait	Idle waiting for I/O
%irq / %soft	Hard / soft interrupts
%steal	Time stolen by hypervisor
%guest	Running guest VMs (host)
%idle	Idle not waiting on I/O

Patterns:

Observation	Likely story
One CPU %usr 100, others idle	Single-thread bottleneck / GIL / hot lock
All CPUs high %usr	Healthy parallel load or capacity shortfall
High %soft on few CPUs	Network/softirq concentration — RSS/RPS/IRQ affinity
High %steal	Noisy neighbor or oversubscribed VM
High %iowait	Storage latency (confirm with <code>iostat -xz</code>)
High %sys	Kernel-heavy workload, syscalls, networking, virtualization

First sample without care about “since boot” applies less than `iostat`, but interval mode is still the right way to watch live change.

Notes & Pitfalls

- Install `sysstat` if missing: `sudo apt install sysstat`.
- CPU numbering is logical CPUs (includes SMT/hyperthreads).
- Inside containers you may still see host CPUs depending on visibility; cgroup limits need `cpuset`/cpu controllers and specialized tools.

- Balancing IRQs is a separate operational topic (`/proc/irq`, `irqbalance`).
- `%iowait` can look “high” on mostly-idle systems with occasional slow I/O — correlate with latency SLOs, not only percentages.

Related Commands

- `vmstat` — compact aggregate loop
- `sar -P ALL` — historical per-CPU
- `top` (key 1) / `htop` — interactive per-CPU
- `pidstat -u` — per-process CPU
- `perf top` — see hot kernel/user symbols
- `nproc` / `lscpu` — how many CPUs exist

Additional Resources

- `man mpstat`
- `man pidstat`

sar

Overview

sar (System Activity Reporter, **sysstat**) prints **live and historical** metrics: CPU, memory, load, disk, network, and more. Live mode is like specialized siblings (**mpstat**, **iostat**); historical mode reads daily binary logs written by **sadc** via cron or systemd timers. On Ubuntu, collection is **off** until enabled.

Syntax

```
sar [options] [interval [count]]
sar [options] -f /var/log/sysstat/saDD
sar [options] -s HH:MM:SS -e HH:MM:SS
```

Common Options

Option	Description
-u	CPU utilization (default)
-r	Memory utilization
-q	Load average / run queue
-b	I/O transfer rates
-d	Per-block device
-n DEV\ EDEV\ TCP\ SOCK\ ...	Network statistics
-P ALL	Per-CPU
-s / -e	Start / end time for historical
-f file	Read a specific saDD data file
-A	Nearly all reports
-o file	Write binary samples (advanced)
-f + date	Files under /var/log/sysstat/ (sa07 = day 07)

Key Use Cases

1. Live sampling during an incident (same family as mpstat/iostat)
2. “What happened at 09:15 yesterday?” from sa logs

3. Capacity trends when full Prometheus stack is overkill
4. Correlate CPU/mem/net windows after outages

Examples with Explanations

Live CPU samples

```
sar -u 1 5
sar -u ALL 1 5
```

Memory and load

```
sar -r 1 5
sar -q 1 5
sar -r -q 1 5
```

Disk and I/O rates

```
sar -b 1 5
sar -d 1 5
```

Network devices

```
sar -n DEV 1 5
sar -n EDEV 1 5    # errors/drops
sar -n TCP,ETCP 1 5
```

Per-CPU history of “now”

```
sar -P ALL 1 3
```

Today’s history (collection must be enabled)

```
sar -u
sar -r
sar -q
ls -la /var/log/sysstat/
```

Without data files, sar only works in live interval mode or errors on historical queries.

Slice a time window

```
sar -u -s 09:00:00 -e 10:00:00
sar -r -s 09:00:00 -e 10:00:00
sar -n DEV -s 09:00:00 -e 10:00:00
```

Read another day's file

```
# Day-of-month file name (locale/sysstat config can vary)
sudo ls /var/log/sysstat/
sar -u -f /var/log/sysstat/sa08
sar -r -f /var/log/sysstat/sa08 -s 14:00:00 -e 15:00:00
```

Enable collection on Ubuntu (sketch)

```
sudo apt install sysstat
# Debian/Ubuntu: enable data collector
sudo sed -i 's/^ENABLED=.* /ENABLED="true" /' /etc/default/sysstat
sudo systemctl enable --now sysstat
# timers: sysstat-collect.timer / sysstat-summary.timer (names may vary by release)
systemctl list-timers | grep -i sysstat
ls /var/log/sysstat/
```

Confirm files appear under `/var/log/sysstat/` after the next collect interval (often ~10 minutes).

Incident bundle

```
sar -u 1 5
sar -r 1 5
sar -q 1 5
sar -n DEV 1 5
# if history enabled:
sar -u -s 11:00:00 -e 12:00:00
```

Understanding Output

Output is tabular with a timestamp column in historical mode. Field meanings align with other sysstat tools:

Report	Useful fields
-u	%user, %system, %iowait, %steal, %idle

Report	Useful fields
-r	kmemfree, kbavail, %memused, kbcached, %swpused
-q	runq-sz, plist-sz, ldavg-1/5/15
-d	tps, rkB/s, wkB/s, await, %util (version-dependent)
-n DEV	rxkB/s, txkB/s, rxpck/s, txpck/s

Reading historical rows: averages between sample points (e.g. every 10 minutes) smooth spikes — a 30-second outage may barely move a 10-minute bucket. For micro-incidents, live ls tools win; sar history wins for longer windows.

Notes & Pitfalls

- **Ubuntu default:** package installed collection enabled — set `ENABLED="true"` and start `sysstat` timers.
- Data retention/rotation is configured under `/etc/sysstat/` and `logrotate` — don't assume months of history.
- Timezone of timestamps follows system clock — wrong TZ confuses windows (`timedatectl`).
- Binary `sa` files are not plain text; always use `sar -f`.
- Modern long-term monitoring is often Prometheus/node_exporter/Grafana; `sar` remains excellent for local/air-gapped hosts.
- Permissions: reading some logs may need root depending on directory mode.

Related Commands

- `vmstat` / `iostat` / `mpstat` — live siblings
- `pidstat` — per-process over intervals
- `journalctl` — logs, not metrics
- `prometheus` / `node_exporter` — long-term modern stacks
- `dstat` / `atop` — alternative live dashboards (if installed)

Additional Resources

- `man sar`
- `man sadc`
- `/usr/share/doc/sysstat/` on package install

lsof

Overview

lsof lists **open files**. On Linux, “files” include regular files, directories, block/char devices, pipes, and **sockets** — so it answers both “what files does this process hold?” and “who has this port?”. Essential for busy **umount**, deleted-but-open disk space mysteries, and classic port ownership checks (modern alternative for sockets alone: **ss -lntp**).

Syntax

```
lsof [options]
lsof -p PID
lsof -i :port
```

Common Options

Option	Description
-p PID	Files for process (comma-separated PIDs)
-i	Internet sockets; optional address/port filter
-iTCP:443 -sTCP:LISTEN	Listening TCP on 443
-u user	Files for user
-c name	Command name prefix match
+D dir	Recurse directory (expensive)
+f -- path	Path as filesystem (busy mount checks)
-n	No host DNS resolution (faster)
-P	No port name resolution (show numbers)
-a	AND selections (default is often OR — easy footgun)
-t	PIDs only (for scripts/kill pipelines)
-R	Show parent PID

Key Use Cases

1. Who is listening on a TCP/UDP port?
2. Why is umount busy?

3. Deleted files still holding disk space
4. Audit files/sockets for a PID or user

Examples with Explanations

Who listens on port 22?

```
sudo lsof -nP -iTCP:22 -sTCP:LISTEN
# modern alternative focused on sockets:
ss -lntp | grep ':22'
```

-nP skips DNS and service names for speed and stable output.

Open files for a PID

```
sudo lsof -p "$(pgrep -x nginx | head -1)"
sudo lsof -p 1234 -nP
```

Configs, logs, .so mappings, and sockets held by that process.

Deleted files still open (disk full mystery)

```
sudo lsof +L1
sudo lsof -nP | grep '(deleted)'
# focus large ones:
sudo lsof +L1 | awk 'NR==1 || $7 ~ /[0-9]/ {print}' | head
```

A process can keep a deleted log open — df shows full, du does not account for it until restart/close.

Files under a mount (before umount)

```
sudo lsof +f -- /mnt/data
sudo lsof /var
sudo fuser -vm /mnt/data
```

Identify shells (cwd), daemons, and lockers preventing umount.

Network connections for a user

```
sudo lsof -u deploy -a -i
sudo lsof -a -u deploy -iTCP -sTCP:ESTABLISHED -nP
```

-a ANDs the filters — without it, -u and -i can OR and surprise you.

Command name filter

```
sudo lsof -c python -nP
sudo lsof -c java -i -nP
```

Prefix match on the command column.

PIDs only for scripting

```
sudo lsof -t -iTCP:8080 -sTCP:LISTEN
# careful:
# sudo kill "$(sudo lsof -t -iTCP:8080 -sTCP:LISTEN)"
```

UNIX sockets / pipes (advanced)

```
sudo lsof -U | head
sudo lsof -p PID -a -U
```

Useful for local agent debugging (docker.sock, etc.).

Understanding Output

Typical columns:

Column	Meaning
COMMAND	Process name
PID	Process ID
USER	Owner
FD	File descriptor (cwd, txt, mem, 0u, 1w, 2u, DEL, ...)
TYPE	REG, DIR, CHR, FIFO, IPv4, IPv6, unix, ...
DEVICE	Device numbers
SIZE/OFF	Size or offset
NODE	Inode / protocol node
NAME	Path, socket endpoint, or (deleted)

FD suffixes: **r** read, **w** write, **u** read/write. **cwd** is current working directory — a common umount blocker when your shell sits on the mount.

Notes & Pitfalls

- Root is required for other users' processes and many sockets.
- Full-system **lsof** is expensive on busy hosts — narrow with **-p**, **-i**, **-u**, paths.
- Default combination of tests can be **OR**; use **-a** when you mean AND.
- Prefer **ss -lntp** for “what listens where?” when you only care about sockets.
- NFS and FUSE paths can hang **lsof** if the remote is stuck — try scoped queries first.
- Containers: run **lsof** in the same PID/mount namespace you care about (host vs container).

Related Commands

- **ss** — modern socket statistics
- **fuser** — PIDs using a file, mount, or port
- **strace** — syscall-level open/connect trail
- **lsof** vs **netstat** — prefer **ss** for sockets-only
- **fuser -vm** / **findmnt** — umount busy workflow
- **ps** / **pgrep** — find PIDs before **lsof -p**

Additional Resources

- **man lsof**
- **man ss**
- **man fuser**

strace

Overview

strace traces **system calls** and signals for a process. When a program fails with a vague error (“permission denied”, “connection refused”, silent hang), strace often shows the failing **openat**, **connect**, **execve**, or **access**. It is a diagnostic scalpel — powerful, noisy, and can slow production processes. Prefer filtering (**-e**) and short attaches.

Syntax

```
strace [options] command [args...]  
strace [options] -p PID
```

Common Options

Option	Description
-e	Syscall sets
trace=file\ network\ process\ desc\ ...	
-e openat,connect,execve	Specific calls
-e errno=ENOENT	Only failures of a type (version-dependent usage)
-o file	Write trace to file (keep stderr clean)
-f	Follow forks/threads (usually want this)
-p PID	Attach to running process
-c	Summary counts and time per syscall
-C	Summary plus regular trace
-s N	Max string length printed (default often 32)
-y	Resolve FD paths
-yy	Extra protocol details for sockets (newer)
-tt / -ttt	Timestamps (wall / epoch)
-T	Time spent in each syscall
-q	Quieter attach/detach messages

Safety

- Attaching to live production processes **adds overhead** — can stall latency-sensitive apps.
- Needs privileges for other users' PIDs (CAP_SYS_PTRACE / root); Yama `ptrace_scope` may block attaches.
- Do not casually attach to security-sensitive processes (agents, secret managers) on shared notes/screens.
- Detach with Ctrl-C carefully; prefer `-o` logs over flooding terminals in incidents.

Key Use Cases

1. Why did a command fail to open a file or config?
2. Where is a hang — blocking `read/poll/futex/connect`?
3. Which addresses/ports does a binary hit?
4. Rough syscall cost profile (`-c`) before deeper `perf`

Examples with Explanations

Why did it fail? (file open)

```
strace -f -e openat,access ls /no/such 2>&1 | tail
strace -f -e openat,stat,access cat /etc/shadow
```

Look for `= -1 ENOENT` or `EACCES` on the path you expected.

Network path of a client

```
strace -f -e connect,sendto,recvfrom,poll,select \
curl -sS https://example.com/ -o /dev/null
```

See DNS vs connect failures (pair with `curl -v / dig`).

Summary profile

```
strace -c ls -R /etc >/dev/null
strace -c -f make -j"${nproc}" >/dev/null
```

Counts and rough time per syscall — good for “too many small reads” stories.

Attach to a running PID

```
pidof myapp
sudo strace -p "$(pidof -s myapp)" -f -e file -s 200
# or:
sudo strace -p PID -f -e trace=network -tt
```

-f follows worker threads. Scope with -e or the terminal becomes unreadable.

Write a full log

```
strace -f -tt -T -o /tmp/trace.txt ./app --flag
less /tmp/trace.txt
grep -E 'ENOENT|EACCES|ECONNREFUSED' /tmp/trace.txt | tail
```

Always useful: keep app stdout free; search the log for errnos.

Longer strings and FD paths

```
strace -f -s 200 -y -e openat,read,write ./app
```

-s avoids truncated paths; -y shows what FD 7 actually is.

Child processes / build systems

```
strace -f -e execve,openat make package 2>&1 | grep execve
```

See which helpers are actually invoked.

Hang diagnosis sketch

```
sudo strace -p PID -f -tt -T
# stuck in futex → locking; poll/select → waiting I/O; read on socket → peer silent
```

Understanding Output

Typical lines:

```
openat(AT_FDCWD, "/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
connect(4, {sa_family=AF_INET, sin_port=htons(443), ...}, 16) = -1 EINPROGRESS (Operation no
```

```
openat(AT_FDCWD, "/etc/foo.conf", O_RDONLY) = -1 ENOENT (No such file or directory)
```

Piece	Meaning
Name	Syscall
Args	Paths, FDs, flags, addresses
= N	Success return (FD number, byte count, ...)
= -1 ERRNO (text)	Failure — usually the smoking gun
Duration (-T)	Time inside the call (blocking shows up here)

Common errnos: ENOENT missing path, EACCES/EPERM rights, ECONNREFUSED nothing listening, ETIMEDOUT network path, EAGAIN nonblocking empty.

Notes & Pitfalls

- Without `-e`, output volume explodes — always filter first when possible.
- Multi-threaded apps need `-f` or you miss workers.
- Go/Java/Node runtimes are syscall-noisy; start with `-e network` or `-e file`.
- `seccomp/sandboxes` may alter available calls; container `seccomp` profiles matter.
- Kernel `yama/ptrace_scope` (`/proc/sys/kernel/yama/ptrace_scope`) can deny non-root attaches.
- For performance at scale prefer `perf`, eBPF (`bpftime`), or app metrics — `strace` is not a profiler for production load tests.

Related Commands

- `ltrace` — library calls (not syscalls)
- `perf` — CPU performance counters / flame graphs
- `gdb` — breakpoints and state inspection
- `lsof` — open files/sockets snapshot without tracing
- `journalctl` / app logs — higher-level errors first
- `ss` / `tcpdump` — network without process tracing

Additional Resources

- `man strace`
- `man ptrace`

perf

Overview

`perf` is Linux’s primary performance analysis suite. It samples hardware performance counters, software events, and kernel tracepoints to show *where* CPU time goes—functions, shared libraries, kernel paths—not just which process is “busy” in `top`.

```
# typical packages
sudo apt install linux-perf          # Debian/Ubuntu (name varies by kernel)
sudo dnf install perf
# ensure matching kernel headers/debug symbols for readable stacks when possible
```

Syntax

```
perf [subcommand] [options] [--] [command]
```

Common Subcommands

Subcommand	Description
<code>stat</code>	Aggregate counters for a command or system-wide
<code>record</code>	Sample events → <code>perf.data</code>
<code>report</code>	Interactive/text report from <code>perf.data</code>
<code>top</code>	Live function-level top
<code>script</code>	Dump raw samples (for flame graphs, custom tools)
<code>list</code>	List available events
<code>annotate</code>	Source/asm annotation
<code>probe</code>	Dynamic probes (advanced)

Common Options (record / stat)

Option	Description
<code>-e EVENT</code>	Event(s): cycles, instructions, cache-misses, ...

Option	Description
<code>-g / --call-graph</code>	Call graphs (<code>fp</code> , <code>dwarf</code> , <code>lbr</code> depending on arch)
<code>-p PID / -t TID</code>	Attach to process/thread
<code>-a</code>	System-wide
<code>-F freq</code>	Sample frequency
<code>-o file</code>	Output file (default <code>perf.data</code>)

Key Use Cases

1. Find hot functions in user or kernel space
2. Measure IPC, cache misses, branch misses
3. System-wide profiling under load
4. Generate data for flame graphs

Examples with Explanations

High-level counters for one command

```
perf stat ./my_program
perf stat -e cycles,instructions,cache-misses,branch-misses ./my_program
```

`instructions / cycles` → rough IPC. High cache-miss rates often explain “CPU busy but slow” workloads.

Live profiling

```
sudo perf top
sudo perf top -p $(pgrep -n myapp)
```

Like `top`, but for symbols. Needs root or `kernel.perf_event_paranoid` relaxed appropriately.

Record + report workflow

```
perf record -g ./my_program
perf report
# or
perf record -F 99 -g -p $(pgrep -n myapp) -- sleep 30
perf report --stdio | less
```

`-g` collects call graphs so you see callers, not only leaf functions.

System-wide snapshot

```
sudo perf record -a -g -- sleep 10
sudo perf report
```

One-liner recipes

```
# Why is this box busy?
sudo perf top -g

# Count context switches / migrations during a run
perf stat -e context-switches,cpu-migrations,page-faults ./server

# Dump samples for external flamegraph tooling
perf script > out.perf
```

Paranoid settings (understand before changing)

```
sysctl kernel.perf_event_paranoid
# -1: least restricted; higher values limit unprivileged use
# Prefer temporary, documented changes over permanent wide-open settings
```

Notes & Pitfalls

- **Symbols:** stripped binaries show hex addresses. Install debuginfo / build with frame pointers or use DWARF call graphs (`--call-graph dwarf`, higher overhead).
- **Overhead:** high-frequency sampling and DWARF unwind cost CPU and disk.
- **Containers:** host `perf` can profile containers with care; cgroup modes and symbol paths complicate analysis.
- **Permissions:** modern kernels gate unprivileged `perf`; don't casually set `paranoid` to `-1` on multi-tenant hosts.
- `perf.data` is bulky—delete or compress after analysis.

2026-relevant notes

- Prefer **frame pointers** (`-fno-omit-frame-pointer` / modern defaults on some stacks) or **ORC/DWARF** unwind as appropriate for your distro/arch.
- Pair with `bpfttrace`, `bcc`, and `pidstat/perf stat` for layered diagnosis: counters → samples → targeted traces.
- For continuous profiling in production, evaluate eBPF-based agents; use classic `perf` for deep dives and offline `perf report`.

Comparison to alternatives

Tool	Role
<code>perf</code>	Sampling + counters, deep Linux integration
<code>strace</code>	Syscall trace (different question)
<code>top</code> / <code>htop</code> / <code>bttop</code>	Process-level overview
<code>bpftrace</code>	Ad-hoc kernel/user probes

Related Commands

- `strace` — syscalls
- `top` / `htop` / `bttop` — process monitors
- `vmstat` / `iostat` / `mpstat` — subsystem stats
- `sysctl` — kernel knobs including perf policy

Additional Resources

- `man perf` and `man perf-record`
- Kernel `tools/perf` documentation
- Brendan Gregg's Linux performance materials (flame graphs, methodology)

btop

Overview

btop is a modern, interactive resource monitor with a polished TUI: CPU (often per-core with clocks/temps when available), memory, disks, network, and a process list. It sits in the same family as htop and bashtop/bpytop, with a denser visual layout.

```
sudo apt install btop      # Debian/Ubuntu when packaged
sudo dnf install btop
# or install from upstream releases if your distro lags
```

Syntax

```
btop [options]
```

Common Options

Option	Description
-lc, --low-color	16-color / limited terminals
-t, --tty	Force tty-friendly mode (check --help on your build)
--utf-force	Force UTF-8 drawing
-p, --preset N	Start with layout preset
-u, --update MS	Refresh rate in milliseconds
-h, --help	Help

Exact flags vary slightly by version—run `btop -h`.

Interactive Keybindings (typical)

Key	Action
↑ ↓ / j k	Move process selection
Enter	Process details / options

Key	Action
f	Filter processes
m	Memory-related layout toggle
p	Process sort / panel focus (version-dependent)
k or signal menu	Kill / send signal to selected process
t	Tree view (if enabled in version)
FF2 / preset keys	Cycle presets
Esc / q	Back / quit
? or h	Help overlay

If a key does nothing, open the in-app help—bindings are configurable.

Key Use Cases

1. At-a-glance host health on a jump box
2. Spot CPU steal, memory pressure, or disk saturation visually
3. Find and signal runaway processes without remembering `ps` pipelines
4. Live network throughput per interface

Examples with Explanations

Start defaults

```
btop
```

Low-color SSH / serial console

```
btop --low-color
# or
btop -lc
```

Faster or slower refresh

```
btop -u 500
btop -u 2000
```

One-liner recipes

```
# Prefer btop when available, else htop, else top
command -v btop >/dev/null && btop || command -v htop >/dev/null && htop || top

# Record a note of load while viewing (separate terminal)
while true; do printf '%s %s\n' "$(date -Iseconds)" "$(cat /proc/loadavg)"; sleep 5; done
```

Notes & Pitfalls

- **Remote SSH:** needs a reasonable terminal (xterm-256color, width). Tiny terminals break layout—use `-lc` or `resize`.
- **Sensors/temps:** may need `lm-sensors` configured; containers often lack host sensors.
- **Permissions:** process details and some disk stats are richer as root but daily use as normal user is fine.
- **Not a logger:** for historical metrics use Prometheus node exporter, Netdata, or `sar`.
- Config usually lives under `~/.config/btop/`—back it up if you customize themes heavily.

2026-relevant notes

- `btop` is a strong default **interactive** monitor; keep `htop` for muscle memory and `top` for universal availability on rescue images.
- On servers, pair TUI tools with `journalctl -f`, `ss -lntp`, and `dmesg -T` rather than expecting one dashboard to explain outages.
- In Kubernetes/node shells, metrics may reflect the container cgroup, not the whole node—interpret carefully.

Comparison to alternatives

Tool	Notes
<code>btop</code>	Modern TUI, visual meters
<code>htop</code>	Ubiquitous interactive process focus
<code>top</code>	Always there, minimal
<code>glances</code>	Cross-platform, optional web UI
<code>btm</code> / <code>zenith</code>	Other modern Rust/Go monitors

Related Commands

- `top` / `htop` — process monitors
- `vmstat` / `iostat` / `mpstat` — batch subsystem stats

- `nproc` / `free` / `df` — quick non-interactive checks
- `perf top` — function-level CPU profiling

Additional Resources

- Upstream `btop` README for install and themes
- `man btop` when packaged

ltrace

Overview

`ltrace` intercepts and prints **dynamic library calls** of a process (similar to how `strace` prints syscalls). Useful when you need to see `malloc`, `strcmp`, `fopen`, or library-level behavior without a full debugger.

```
sudo apt install ltrace
```

Syntax

```
ltrace [options] command [args...]  
ltrace -p PID
```

Common Options

Option	Description
<code>-e filter</code>	Include/exclude symbols
<code>-p PID</code>	Attach to process
<code>-c</code>	Counts / summary
<code>-T</code>	Show time in calls
<code>-f</code>	Follow forks
<code>-o file</code>	Output file
<code>-s strsize</code>	String capture length

Safety

- Attaching can slow or perturb production processes.
- Needs appropriate `ptrace` permissions (`kernel.yama.ptrace_scope`).
- Do not leave high-volume tracing on busy services.

Examples with Explanations

Trace a command

```
ltrace -e fopen+fclose+read ls /tmp 2>&1 | head
```

Summary counts

```
ltrace -c true  
ltrace -c ./myapp --flag 2>&1 | tail
```

Attach

```
ltrace -p "$(pidof myapp)" -e 'malloc+free' -c
```

Notes & Pitfalls

- Statically linked binaries show little.
- Prefer **strace** for syscall/ENOENT path issues; **ltrace** for library logic.
- Go/Rust binaries may be less informative depending on linkage.

Related Commands

- **strace** — syscalls
- **gdb** — full debugger
- **perf** — profiling
- **nm** / **ldd** — symbols and libraries

Additional Resources

- `man ltrace`

Users and Groups

Intro

Intro

Create and modify accounts, password aging, groups, and privilege elevation. Prefer `sudo` for policy-controlled admin work; use account tools carefully on production identity stores.

Commands in this part

Command	Role
<code>sudo</code>	<code>sudo</code> runs a command as another user (default: root) according to policy in <code>/etc/sudoers</code> and <code>/etc/sudoers.d/*</code> .
<code>visudo</code>	<code>visudo</code> safely edits the <code>sudoers</code> configuration (<code>/etc/sudoers</code> and files under <code>/etc/sudoers.d/</code>).
<code>useradd</code>	<code>useradd</code> creates a new user account and related system entries (<code>/etc/passwd</code> , <code>/etc/shadow</code> , <code>/etc/group</code> as needed).
<code>usermod</code>	<code>usermod</code> modifies an existing user account: shell, home, groups, UID, lock state, expiry, and more.
<code>userdel</code>	<code>userdel</code> deletes a user account from the system.
<code>passwd</code>	<code>passwd</code> sets or changes user passwords and can lock/unlock accounts or expire passwords.
<code>chage</code>	<code>chage</code> (change age) views and modifies password aging policy for a user: last change date, minimum/maximum days...
<code>groupadd</code>	<code>groupadd</code> creates a new group entry in the local group database (<code>/etc/group</code> , and <code>/etc/gshadow</code> when used).
<code>groupmod</code>	<code>groupmod</code> modifies an existing group: rename it, change its GID, or adjust related attributes depending on the...
<code>groupdel</code>	<code>groupdel</code> deletes a group from the system group database (typically <code>/etc/group</code> and <code>/etc/gshadow</code>).

Command	Role
<code>getent</code>	<code>getent</code> queries Name Service Switch (NSS) databases—the same view applications get via <code>glibc</code> : local files, LDAP/SSSD,...
<code>su</code>	<code>su</code> (“substitute user”) starts a shell or runs a command as another user.

Suggested starting points

1. Elevation: `sudo`, `visudo`, `su`.
2. Accounts: `useradd`/`usermod`/`userdel`, `passwd`, `chage`.
3. Groups: `groupadd`/`groupmod`/`groupdel`.
4. Lookups: `getent` (works with NSS/LDAP backends, not only local files).

Related parts

- Security — SELinux contexts when access still fails
- Files and paths — ownership and modes
- Services and runtime — run daemons as dedicated users

Continue with the individual command pages in this part.

sudo

Overview

`sudo` runs a command as another user (default: **root**) according to policy in `/etc/sudoers` and `/etc/sudoers.d/*`. Prefer one-shot elevation over lingering in a root shell. On Ubuntu, members of the `sudo` group typically get full rights via the distro `sudoers` snippet.

Syntax

```
sudo [options] [command]
sudo -i
sudo -u user command
sudoedit file           # same as sudo -e
```

Common Options

Option	Description
<code>-u user</code>	Run as user (not root)
<code>-g group</code>	Set primary group for the command
<code>-i</code>	Login shell as target user (loads target profile)
<code>-s</code>	Shell as target user (minimal env change)
<code>-E</code>	Preserve environment (policy permitting — often restricted)
<code>-k</code>	Invalidate cached credentials
<code>-K</code>	Remove timestamp entirely
<code>-l / -l -U user</code>	List allowed commands
<code>-n</code>	Non-interactive (fail if a password would be needed)
<code>-v</code>	Validate / extend timestamp without running a command
<code>-e / sudoedit</code>	Edit file safely as root
<code>-b</code>	Run command in background
<code>-g group</code>	Primary group for the command

Key Use Cases

1. Administrative package and service changes
2. Least-privilege elevation per command
3. Audited privileged actions (auth log / journal)
4. Safe file edits as root (`sudoedit`)
5. Run maintenance as a service account (`-u`)

Safety

- **Never** `chmod` sudoers to world-writable. Always edit with `visudo`.
- Pipelines: `sudo cat file | grep x` only elevates `cat`. Use `sudo sh -c '...'` or `sudo grep x file` when the whole pipeline needs root.
- Avoid `sudo su` / `sudo -i` as a daily habit — harder to audit, easy to leave open.
- Be careful with `sudo -E` and `env_keep` — leaking `LD_PRELOAD`/`PATH` into root is a classic privilege path.
- `sudoedit` uses your editor on a temp copy; do not point `$EDITOR` at untrusted wrappers.

Examples with Explanations

Run one admin command

```
sudo apt update
sudo systemctl restart nginx
```

Elevates only those processes; your shell remains an unprivileged user.

Edit a system file safely

```
sudoedit /etc/hosts
sudo -e /etc/ssh/sshd_config
```

Writes to a temporary file and installs only if the editor exits successfully — safer than some `sudo vim` permission edge cases.

Root login shell when you need many steps

```
sudo -i
# ... work ...
exit
```

Loads root's login environment (HOME=/root, profile scripts). Exit promptly when finished.

Run as another service user

```
sudo -u www-data ls -la /var/www
sudo -u postgres psql -c '\conninfo'
```

Debug permissions and DB access as the account that actually runs the service.

See your privileges

```
sudo -l
```

Shows what policy allows for your account (and matching host/command rules).

Pass env vars explicitly (preferred over -E)

```
sudo ENV=prod /usr/local/bin/deploy
sudo --preserve-env=http_proxy,https_proxy apt update # when policy allows
```

Prefer naming variables over blanket environment preservation.

Credential cache control

```
sudo -v          # refresh timestamp
sudo -k          # drop cache for this tty (next sudo asks again)
```

Default timeout is often ~15 minutes per tty — do not assume it on shared jump hosts.

Non-interactive / scripts

```
sudo -n true || { echo "sudo needs a password; cannot continue" >&2; exit 1; }
sudo -n systemctl is-active nginx
```

-n fails instead of prompting — essential for unattended scripts.

Dangerous patterns to avoid

```
# BAD: elevates only the left side
sudo cat /etc/shadow | grep root

# GOOD:
sudo grep root /etc/shadow
sudo sh -c 'cat /etc/shadow | grep root'
```

Read logs of sudo use

```
journalctl -u sudo -n 50 --no-pager          # when unit exists
grep -i sudo /var/log/auth.log | tail        # Debian/Ubuntu rsyslog path
journalctl _COMM=sudo -n 50 --no-pager
```

Understanding Output

Successful commands look like the underlying tool's output. Failures include password rejects, policy denials (Sorry, user ... is not allowed...), and missing tty for password prompts. Exit status is usually that of the invoked command; policy denials return non-zero without running it.

Notes & Pitfalls

- Ubuntu's default is group-based admin (%sudo ALL=(ALL:ALL) ALL) — still prefer explicit command lists on hardened hosts.
- Lecture/insults and password prompts are configured in sudoers (Defaults).
- `sudo !!` re-runs the previous shell command with sudo — review !! expansion before Enter.
- Root-equivalent via misconfigured NOPASSWD: ALL is convenient and dangerous; scope it tightly.
- Containers: passwordless sudo inside images is common for convenience — treat those images as root-equivalent.

Related Commands

- `visudo` — edit sudoers safely

- `su` / `runuser` — switch user (different auth model)
- `passwd` — change passwords
- `pkexec` — polkit elevation on desktops
- `loginctl` — session / lingering control

Additional Resources

- `man sudo` / `man sudoers`
- `visudo(8)`

visudo

Overview

`visudo` safely edits the **sudoers** configuration (`/etc/sudoers` and files under `/etc/sudoers.d/`). It locks the file and **syntax-checks** before installing changes — preventing a bad edit from locking out all sudo access. Always use `visudo` instead of editing sudoers directly with a random editor.

Syntax

```
visudo [options]
visudo -f file
```

Common Options

Option	Description
<code>-f file, --file=file</code>	Edit a specific file (e.g. drop-in)
<code>-c, --check</code>	Check syntax only
<code>-s, --strict</code>	Stricter parse
<code>-q, --quiet</code>	Quiet
<code>-x file</code>	Export as JSON (newer) / legacy dump options vary
<code>-N</code>	No default includes (see man)

Editor from `$EDITOR` / `$VISUAL` / compile-time default (`vi`/`nano`).

Examples with Explanations

Edit main sudoers

```
sudo visudo
```


Edit a drop-in (preferred)

```
sudo visudo -f /etc/sudoers.d/90-alice
```

Drop-ins should be numeric-prefixed and avoid `.` or `~` in names (sudo ignores many invalid names).

Check syntax

```
sudo visudo -c
sudo visudo -cf /etc/sudoers.d/90-alice
```

Common policy snippets (examples)

```
# /etc/sudoers.d/90-alice (via visudo -f)
alice ALL=(ALL) ALL
# passwordless for a single command (still risky):
alice ALL=(root) NOPASSWD: /usr/bin/systemctl restart nginx
# group based:
%sudo ALL=(ALL:ALL) ALL
```

Understand tag implications (NOPASSWD, SETENV, NOEXEC) before deploying.

Recover from broken sudoers (prevention)

If you never skip `visudo`, you avoid the classic lockout. Recovery usually needs:

- console/root login
- or break-glass cloud serial console
- then `visudo` fix

Test as user

```
sudo -l -U alice
sudo -u alice sudo -l
```

Notes / Pitfalls

- Never `nano /etc/sudoers` without `visudo` — a typo can deny all sudo.
- Last matching rule wins in many cases — order matters.
- `#includedir /etc/sudoers.d` is standard; file names must meet sudo's filters.
- `NOPASSWD: ALL` is almost always too broad for humans.
- Keep a root console session open while testing policy changes on remote hosts.

2026-relevant notes

- Prefer small drop-ins per role over one monolithic sudoers file.
- Many orgs move to SSH certs + policy engines; sudoers still ubiquitous on servers.
- Audit with `sudo -l` and central config management (Ansible `template` + `validate`).

Related Commands

- `sudo / sudo -l`
- `sudoedit` — edit files as another user safely
- `pkexec` — polkit privilege (desktop/admin)
- `su` — switch user
- editor env: `EDITOR=vim visudo`

Additional Resources

- `man visudo`, `man sudoers`

useradd

Overview

`useradd` creates a new user account and related system entries (`/etc/passwd`, `/etc/shadow`, `/etc/group` as needed). On **Ubuntu/Debian**, the friendlier interactive wrapper is **`adduser`**; `useradd` is the low-level, script-friendly tool common in configuration management. Creating a login-ready human usually means: `useradd -m` → `passwd` → optional groups → verify with `getent/id`.

Syntax

```
sudo useradd [options] LOGIN
```

Common Options

Option	Description
<code>-m</code> , <code>--create-home</code>	Create home directory from skeleton
<code>-M</code>	Do not create home (override defaults)
<code>-d HOME</code> , <code>--home-dir</code>	Home path (default under <code>/home</code>)
<code>-s SHELL</code> , <code>--shell</code>	Login shell
<code>-g GROUP</code> , <code>--gid</code>	Primary group (name or GID)
<code>-G GROUPS</code> , <code>--groups</code>	Comma-separated supplementary groups
<code>-u UID</code> , <code>--uid</code>	Numeric UID
<code>-c COMMENT</code> , <code>--comment</code>	GECOS (full name, room, ...)
<code>-r</code> , <code>--system</code>	System account (UID from system range; often no aging)
<code>-e EXPIRE</code> , <code>--expiredate</code>	Account expiry (YYYY-MM-DD)
<code>-f INACTIVE</code>	Days after password expiry until disable
<code>-p PASSWORD</code>	Encrypted password string (prefer <code>passwd/chpasswd</code>)
<code>-k SKEL</code>	Skeleton directory (default <code>/etc/skel</code>)
<code>-N</code>	Do not create a group with the same name as the user
<code>-U</code>	Create a same-named user-private group (common default)
<code>-D</code>	Display (or set, with flags) default <code>useradd</code> values

Defaults and files

Path	Role
/etc/login.defs	UID/GID ranges, mail dir, umask-related defaults
/etc/default/useradd	Default shell, home base, expire, skeleton
/etc/skel/	Files copied into new homes with <code>-m</code>
/etc/passwd / /etc/shadow / /etc/group	Account databases

View effective defaults:

```
sudo useradd -D
```

Safety

- Do not pass cleartext secrets on the command line via `-p` — they land in shell history and process listings; use `passwd` or carefully controlled `chpasswd`.
- Choose UIDs deliberately when syncing NFS/LDAP identities — collisions break ownership.
- System accounts (`-r`) should usually get `/usr/sbin/nologin` or `/bin/false`.
- Creating users with `sudo/admin` groups is a privilege decision — prefer documented role groups.

Key Use Cases

1. Provision local human accounts on Ubuntu servers
2. Create service accounts for daemons
3. Align UID/GID with networked filesystems
4. Script reproducible account creation in labs/CI images

Examples with Explanations

Example: human user with home and bash

```
sudo useradd -m -s /bin/bash -c 'Alice Admin' alice
sudo passwd alice
id alice
getent passwd alice
ls -la /home/alice
```

`-m` is easy to forget — without it, no home directory is created on many defaults.

Example: Ubuntu interactive alternative

```
sudo adduser charlie
```

Walks through password and GECOS prompts; preferred for one-off humans on Debian/Ubuntu.

Example: supplementary groups at creation

```
sudo useradd -m -s /bin/bash -G sudo,docker bob
id bob
```

`-G` sets supplementary groups. On Ubuntu, membership in `sudo` grants sudo via the default sudoers rule (still verify policy).

Example: service / system account

```
sudo useradd -r -s /usr/sbin/nologin -d /nonexistent -c 'App service' appsvc
getent passwd appsvc
```

No interactive shell; often no home. Pair with systemd `User=` directives.

Example: fixed UID/GID for NFS

```
sudo groupadd -g 12000 appdata
sudo useradd -m -u 12000 -g appdata -s /bin/bash appuser
id appuser
```

Match numeric IDs across servers that share filesystems without centralized identity.

Example: account expiry

```
sudo useradd -m -e 2026-12-31 -s /bin/bash contractor
sudo chage -l contractor
```

Expiry disables login after the date; refine aging with `chage`.

Example: custom home location

```
sudo useradd -m -d /srv/homes/alice -s /bin/bash alice
```

Ensure parent directories exist and permissions make sense before login.

Example: do not create user-private group

```
sudo useradd -m -N -g users -s /bin/bash dave
```

-N avoids a per-user group when site policy uses a shared primary group.

Example: verify skeleton contents

```
ls -la /etc/skel
sudo useradd -m -k /etc/skel -s /bin/bash skeltest
ls -la /home/skeltest
sudo userdel -r skeltest
```

Custom skeletons help place mandatory `.bashrc` snippets or ssh config templates.

Notes & Pitfalls

- **useradd vs adduser:** different tools; scripts should call `useradd` for stable flags; humans on Ubuntu often use `adduser`.
- Without `-m`, applications expecting `$HOME` fail in confusing ways.
- Primary group (`-g`) vs supplementary (`-G`): both matter for file access.
- LDAP/FreeIPA/sss environments: create users in the identity system, not only locally — local-only accounts confuse operators.
- Removing accounts is `userdel` (see that page); home cleanup needs `-r` carefully.
- Default shell paths differ: `/bin/bash` vs `/usr/bin/bash` — use paths present on the image (`getent passwd` after creation).

Related Commands

- `adduser` — Debian/Ubuntu interactive helper
- `usermod` — modify existing accounts
- `userdel` — delete accounts
- `groupadd` / `gpasswd` — groups
- `passwd` / `chage` — passwords and aging
- `getent` / `id` — verify lookups

Additional Resources

- `man useradd`
- `man login.defs`
- `man adduser` (Debian/Ubuntu)

usermod

Overview

`usermod` modifies an existing user account: shell, home, groups, UID, lock state, expiry, and more. Prefer targeted flags over hand-editing `/etc/passwd`. For group membership, know the difference between **replace** (`-G`) and **append** (`-aG`) — forgetting `-a` is a classic outage that drops sudo rights.

Syntax

```
sudo usermod [options] LOGIN
```

Common Options

Option	Description
<code>-l NEWNAME, --login</code>	Rename login
<code>-c COMMENT, --comment</code>	GECOS field
<code>-d HOME, --home</code>	New home path
<code>-m, --move-home</code>	Move home contents when used with <code>-d</code>
<code>-s SHELL, --shell</code>	Login shell
<code>-u UID, --uid</code>	Change UID (files not auto-chowned everywhere)
<code>-g GROUP, --gid</code>	Primary group
<code>-G GROUPS, --groups</code>	Set supplementary groups to this list (replaces!)
<code>-a, --append</code>	With <code>-G</code> , append groups instead of replacing
<code>-L, --lock</code>	Lock password
<code>-U, --unlock</code>	Unlock password
<code>-e DATE, --expiredate</code>	Account expiry (YYYY-MM-DD or empty to clear)
<code>-f DAYS, --inactive</code>	Inactivity after password expiry
<code>-p PASSWORD</code>	Set encrypted password (prefer <code>passwd</code>)
<code>-r</code>	When changing UID, experimental options vary — read man page carefully

Critical group flags: always use `-aG` to add supplementary groups without wiping existing ones.

Safety

- **usermod -G group user without -a replaces all supplementary groups.** Operators have locked themselves out of `sudo` this way. Prefer:

```
sudo usermod -aG docker alice
```

- Changing UID/home on a live logged-in user confuses running sessions; schedule maintenance.
- Locking (-L) blocks password auth; SSH keys may still work — know your access paths.
- Rename (-1) does not rewrite crontabs, systemd units, mail spools, or ACLs automatically.

Key Use Cases

1. Add a user to `sudo`, `docker`, or app groups (**-aG**)
2. Change login shell or GECOS
3. Move/rename home during account cleanup
4. Lock contractors without deleting data immediately

Examples with Explanations

Example: append supplementary groups (-aG)

```
id alice
sudo usermod -aG docker,sudo alice
id alice
# alice must re-login (or newgrp) for group membership to apply fully
```

-aG is the safe daily form. Verify with `id` after a fresh login.

Example: wrong vs right group edit

```
# DANGEROUS - replaces supplementary groups with only "docker"
sudo usermod -G docker alice

# CORRECT - append docker, keep sudo and others
sudo usermod -aG docker alice
```

If you truly intend to set the definitive list, pass the full list explicitly:

```
sudo usermod -G sudo,docker,adm alice
```


Example: change shell

```
sudo usermod -s /bin/bash alice
getent passwd alice
```

Common fix when an account was created with `nologin` by mistake (for human users only).

Example: lock and unlock

```
sudo usermod -L alice
passwd -S alice
sudo usermod -U alice
```

Related: `passwd -l/-u`. Locking is not the same as setting shell to `nologin` — consider both for service accounts.

Example: move home directory

```
sudo usermod -d /srv/homes/alice -m alice
ls -la /srv/homes/alice
getent passwd alice
```

`-m` moves files from the old home. Ensure destination parent exists; watch disk space and SELinux/AppArmor contexts on hardened systems.

Example: rename login

```
sudo usermod -l newalice alice
# often also:
sudo usermod -d /home/newalice -m newalice
```

Update references in sudoers, cron, CI credentials, and docs.

Example: change primary group

```
sudo groupadd appteam
sudo usermod -g appteam alice
id alice
```

Primary group affects default group of newly created files (with usual `umask`/dir sticky behaviors).

Example: change UID carefully

```
sudo usermod -u 12000 alice
# fix ownership of home (minimum):
sudo chown -R alice:alice /home/alice
# search for old UID leftovers if needed:
sudo find / -xdev -uid 1001 2>/dev/null
```

UID changes do not rewrite the whole filesystem for you.

Example: expiry

```
sudo usermod -e 2026-12-31 alice
sudo usermod -e '' alice          # clear expiry (syntax may vary; confirm man)
sudo chage -l alice
```

Example: service account hardening

```
sudo usermod -s /usr/sbin/nologin -L appsvc
```

No interactive shell + locked password for non-human accounts (SSH keys should also be absent).

Notes & Pitfalls

- Group changes apply to **new** sessions; existing SSH sessions keep old group sets until re-login.
- `newgrp` / `sg` can temporarily switch for testing without full logout.
- Directory services (sssd/LDAP): modify the identity source of truth, not only the local cache.
- `-a` without `-G` is an error; they are a pair for append mode.
- Prefer `gpasswd -a user group` as an alternative for single-group adds.

Related Commands

- `useradd` / `userdel` — create/delete
- `groupadd` / `gpasswd` / `groupmod` — group side
- `passwd` / `chage` — password and aging
- `id` / `getent` — verify
- `vim` — last-resort file edit (avoid)

Additional Resources

- `man usermod`
- `man group(5)`, `man passwd(5)`

userdel

Overview

`userdel` deletes a user account from the system. By default it removes the account entry but may leave the home directory and mail spool unless you pass `-r`. Always confirm you have backups and that no critical processes still run as that user.

Syntax

```
userdel [options] LOGIN
```

Common Options

Option	Description
<code>-r, --remove</code>	Remove home directory and mail spool
<code>-f, --force</code>	Force removal even if logged in / files busy (dangerous)
<code>-Z</code>	Remove SELinux user mapping (when applicable)
<code>-R CHROOT</code>	Chroot
<code>-P PREFIX</code>	Prefix

Examples with Explanations

Delete account, keep home

```
id bob
sudo userdel bob
ls /home/bob          # may still exist
```

Delete account and home

```
sudo userdel -r bob
```

Refuse when logged in

```
who | grep bob
loginctl user-status bob
sudo userdel bob          # may fail if logged in
# terminate sessions first:
sudo loginctl terminate-user bob
sudo userdel -r bob
```

Processes still running

```
ps -u bob
sudo pkill -u bob
sudo userdel -r bob
```

Orphan files outside home

```
# after delete, numeric UID may remain on files:
sudo find /var /srv -nouser 2>/dev/null | head
```

Reassign or delete intentionally.

System users

```
# prefer package uninstall / systemd DynamicUser cleanup
# only manually remove custom system accounts you created
sudo userdel -r appuser
```

Notes / Pitfalls

- Without `-r`, homes accumulate on disk — disk bloat and privacy residue.
- Force-deleting logged-in users can confuse running jobs; terminate cleanly first.
- Crontabs, systemd user units, and mail spools may need manual cleanup.
- UID reuse: deleting then creating a new user with same UID exposes old files — wipe or reown.
- LDAP users: use directory tools, not local `userdel`, for network accounts.

2026-relevant notes

- Prefer lifecycle via configuration management and IdP offboarding playbooks.
- Containers: removing users in a container image layer doesn't rewrite earlier layers' files.
- Check `loginctl`, `crontab -u`, and `/var/lib/systemd/` leftovers for interactive users.

Related Commands

- `useradd` / `usermod`
- `passwd` / `chage`
- `groupdel`
- `find -nouser`
- `loginctl`
- `vi` — manual editor (last resort)

Additional Resources

- `man userdel`

passwd

Overview

passwd sets or changes user passwords and can lock/unlock accounts or expire passwords. Ordinary users change their own password; root can set any user's password and manage lock state. Aging policy is refined with **chage**.

Syntax

```
passwd [options] [LOGIN]
```

Common Options

Option	Description
-S, --status	Account password status
-l, --lock	Lock password (prefix ! in shadow)
-u, --unlock	Unlock password
-d, --delete	Delete password (careful: may allow passwordless login depending on PAM)
-e, --expire	Expire password; force change on next login
-n MIN / -x MAX / -w WARN / -i INACT	Aging shortcuts (often prefer chage)
-a, --all	Status for all users (root)
--stdin	Read password from stdin (root; scripts — prefer chpasswd carefully)

Examples with Explanations

Change passwords

```
passwd          # own password
sudo passwd alice # set alice's password as root
```

Status

```
passwd -S alice
sudo passwd -Sa | head
```

Typical status letters: P (usable password), L (locked), NP (no password) — confirm with `man passwd` on your distro.

Lock / unlock

```
sudo passwd -l alice
sudo passwd -u alice
sudo usermod -L alice      # related
sudo usermod -U alice
```

Locking disables password auth; SSH keys may still work depending on config.

Force change next login

```
sudo passwd -e alice
# or
sudo chage -d 0 alice
```

Scripted set (controlled environments)

```
echo 'alice:NewSecurePass' | sudo chpasswd
# passwd --stdin is not portable across all distros
```

Prefer configuration management vaults over shell history for secrets.

Service accounts

```
sudo passwd -l svc_backup
# or lock shell:
sudo usermod -s /usr/sbin/nologin svc_backup
```


Notes / Pitfalls

- PAM policy enforces complexity/retries — errors may be vague.
- Root bypasses most complexity checks — still use strong secrets.
- Deleting passwords (`-d`) can be dangerous with certain PAM stacks.
- LDAP/sss users: password changes may need `passwd` via SSSD or IdP self-service.
- Shadow file integrity: never edit `/etc/shadow` by hand if tools exist.

2026-relevant notes

- Prefer SSO/IdP for humans; local `passwd` for break-glass and system accounts.
- SSH certificate / key-only logins reduce password surface.
- Pair with `chage` for compliance aging on local accounts.

Related Commands

- `chage` — aging policy
- `chpasswd` — batch updates
- `usermod -L/-U` — lock flags
- `vipw -s` — edit shadow carefully
- `pam-auth-update` — PAM stacks (Debian family)

Additional Resources

- `man passwd`, `man shadow`, `man chage`

chage

Overview

`chage` (change age) views and modifies **password aging** policy for a user: last change date, minimum/maximum days between changes, warn days, and account inactivity/expiry. Complements `passwd` (set password) and is useful for compliance-driven rotation policies.

Syntax

```
chage [options] LOGIN
chage -l LOGIN           # list aging info
```

Common Options

Option	Description
<code>-l, --list</code>	Show aging info
<code>-d DATE, --lastday</code>	Last password change day
<code>-E DATE, --expiredate</code>	Account expiration date (-1 never)
<code>-I DAYS, --inactive</code>	Days after password expiry until account inactive (-1 never)
<code>-m DAYS, --mindays</code>	Minimum days between password changes
<code>-M DAYS, --maxdays</code>	Maximum password age
<code>-W DAYS, --warndays</code>	Warn days before password expiry
<code>-R CHROOT</code>	Chroot

Dates are often YYYY-MM-DD or days since 1970-01-01 depending on option forms — see `man chage`.

Examples with Explanations

View policy

```
sudo chage -l alice
```

Force change on next login

```
sudo chage -d 0 alice
# last day set so password is immediately expired → user must change
```

Set max age 90 days, warn 14

```
sudo chage -M 90 -W 14 alice
```

Disable aging

```
sudo chage -M -1 -E -1 alice
```

Account expiry date

```
sudo chage -E 2026-12-31 contractor
sudo chage -E -1 contractor      # clear expiry
```

Batch service accounts

```
# service accounts often should not expire interactively
sudo chage -M -1 -I -1 -E -1 svc_backup
```

Combine with passwd

```
sudo passwd alice
sudo chage -l alice
```

Notes / Pitfalls

- Requires shadow suite and proper privileges.
- Policies in `/etc/login.defs` supply defaults for new users; `chage` adjusts per-user.
- LDAP/sss environments may store aging in the directory — local `chage` won't apply.
- Forcing `-d 0` on service accounts can break automation that cannot interactively change passwords.
- Date formats and epoch-day forms are easy to mis-set — always `-1` to verify.

2026-relevant notes

- Prefer centralized IdP password policies when users live in SSO; use **chage** for local/system accounts.
- Compliance scanners still look at shadow aging fields on classic Linux servers.
- Pair with **passwd -S** and account locking (**usermod -L**) for full lifecycle control.

Related Commands

- **passwd** — set/lock passwords
- **usermod** — account flags
- **chage -l** vs **passwd -S**
- **vipw / vipw -s** — edit **passwd/shadow** carefully
- **pam_unix / login.defs** — defaults

Additional Resources

- **man chage**, **man shadow**

groupadd

Overview

groupadd creates a new group entry in the local group database (`/etc/group`, and `/etc/gshadow` when used). Groups grant shared file access and sudo/role membership without sharing login accounts. On Ubuntu, interactive admins sometimes use `addgroup`; automation should prefer `groupadd` for stable flags.

Syntax

```
sudo groupadd [options] GROUP
```

Common Options

Option	Description
<code>-g GID, --gid</code>	Numeric GID
<code>-r, --system</code>	System group (GID from system range in <code>login.defs</code>)
<code>-f, --force</code>	Exit successfully if group already exists; still fails on GID conflict unless handled
<code>-K KEY=VAL</code>	Override <code>/etc/login.defs</code> defaults for this invocation
<code>-o, --non-unique</code>	Allow non-unique GID (usually avoid)
<code>-p PASSWORD</code>	Encrypted group password (rare; prefer no group passwords)
<code>-U USERS, --users</code>	Comma-separated user list to add (util-linux; version-dependent)

Related membership tools

Creating a group is only half the job — membership is managed with:

```
sudo usermod -aG GROUP USER
sudo gpasswd -a USER GROUP
sudo gpasswd -d USER GROUP
getent group GROUP
```

```
id USER
```

Safety

- Pick GIDs deliberately when using NFS or shared storage; mismatched GIDs mean “same name, different access”.
- Do not reuse GIDs of deleted groups until you understand leftover file ownership on disk.
- Group passwords are an old pattern; modern sites use explicit membership instead.
- Naming collisions with system groups (`sudo`, `adm`, `disk`, `docker`) are high risk — list first.

Key Use Cases

1. Create application/shared-data groups (`appdata`, `deploy`)
2. Create system groups for daemons
3. Align GIDs across a fleet for shared filesystems
4. Prepare groups before `useradd -G` / `usermod -aG`

Examples with Explanations

Example: simple project group

```
sudo groupadd appdata
getent group appdata
```

Creates the group with the next available GID from the user range.

Example: fixed GID for NFS fleet

```
sudo groupadd -g 12000 appdata
getent group appdata
```

Use the same GID on every server that mounts the same export.

Example: system group for a daemon

```
sudo groupadd -r appsvc
getent group appsvc
```

System GID range is defined in `/etc/login.defs` (`SYS_GID_MIN` / `SYS_GID_MAX`).

Example: create group then add members (-aG)

```
sudo groupadd deploy
sudo usermod -aG deploy alice
sudo usermod -aG deploy bob
getent group deploy
id alice
```

Always **append** with `usermod -aG` so existing supplementary groups remain intact.

Example: Ubuntu interactive alternative

```
sudo addgroup designers
```

Debian/Ubuntu helper; fine for humans, less ideal for exact GID control in scripts.

Example: idempotent-ish create in scripts

```
if ! getent group appdata >/dev/null; then
    sudo groupadd -g 12000 appdata
fi
```

Or experiment with `groupadd -f` carefully — still verify GID when identity matters.

Example: shared directory ownership pattern

```
sudo groupadd appdata
sudo mkdir -p /srv/appdata
sudo chown root:appdata /srv/appdata
sudo chmod 2775 /srv/appdata
# setgid bit (2) → new files inherit group appdata
```

Classic multi-user write directory pattern; pair with sensible umask or ACL if needed.

Example: list and inspect

```
getent group | tail
getent group sudo
grep '^docker:' /etc/group
```

Prefer `getent` so NSS sources (LDAP) are included.

Notes & Pitfalls

- **Name vs number:** tools and NFS care about GID; humans care about names — keep them consistent across systems.
- Deleting groups (`groupdel`) does not rewrite file ownership on disk — orphaned GIDs appear in `ls -ln`.
- Membership changes need re-login to appear in a user's active session.
- Directory services: create groups in FreeIPA/LDAP when that is the source of truth.
- `docker/socket` groups grant near-root power — treat membership as privileged.

Related Commands

- `groupdel` / `groupmod` — delete/rename/change GID
- `gpasswd` — members and administrators of a group
- `usermod -aG` — append user to group
- `useradd -G` — supplementary groups at creation time
- `getent group / id` — verify
- `addgroup` — Debian/Ubuntu helper

Additional Resources

- `man groupadd`
- `man group(5)`
- `man login.defs`

groupmod

Overview

groupmod modifies an existing group: rename it, change its GID, or adjust related attributes depending on the implementation. Membership lists are often managed via **usermod/gpasswd** rather than groupmod.

Requires root privileges.

Syntax

```
groupmod [options] GROUP
```

Common Options

Option	Description
-n NEWNAME, --new-name	Rename group
-g GID, --gid	Change GID
-o, --non-unique	Allow non-unique GID
-p PASSWORD	Set encrypted password (rare; prefer gpasswd)
-R CHROOT	Chroot directory
-P PREFIX	Prefix directory

Examples with Explanations

Rename a group

```
getent group devs
sudo groupmod -n developers devs
getent group developers
```

Change GID

```
getent group developers
sudo groupmod -g 1200 developers
```

Warning: files owned by the old numeric GID are **not** rewritten automatically. Plan a `find/chgrp` migration.

Migrate file ownership after GID change

```
old=1004
new=1200
sudo find /srv/data -group $old -exec chgrp $new {} +
```

Scope carefully; avoid full-filesystem finds without need.

Non-unique GID (rare)

```
sudo groupmod -o -g 1000 shared
```

Usually a bad idea — collisions confuse tools.

Verify

```
getent group developers
grep developers /etc/group /etc/gshadow
```

Notes / Pitfalls

- Renaming may break configs that hard-code the old group name (sudoers, ACLs, unit files).
- Changing GID without file migration leaves orphaned ownership.
- Directory services: local `groupmod` won't change LDAP groups.
- Running processes keep old credentials until re-login / re-spawn.
- Avoid editing `/etc/group` by hand when `groupmod` exists.

2026-relevant notes

- Infrastructure as code should declare desired group names/GIDs immutably; avoid frequent renames.
- Container images: GID consistency across images matters for shared volumes — set explicitly.
- Prefer supplementary group membership management with `usermod -aG` over renaming shared groups.

Related Commands

- `groupadd` / `groupdel`
- `usermod` / `gpasswd`
- `getent group`
- `chgrp`
- `newgidmap` — user namespace gid maps (advanced)

Additional Resources

- `man groupmod`

groupdel

Overview

`groupdel` deletes a group from the system group database (typically `/etc/group` and `/etc/gshadow`). It does **not** delete users who had that group as supplementary — it removes the group entry. You cannot delete a group that is still any user's **primary** group.

Requires root (or equivalent CAP_DAC / account tools).

Syntax

```
groupdel [options] GROUP
```

Common Options

Option	Description
<code>-f, --force</code>	Force deletion in some situations (implementation-dependent; use carefully)
<code>-h, --help</code>	Help
<code>-R, --root CHROOT</code>	Apply in chroot
<code>-P, --prefix PREFIX</code>	Alternate passwd/group prefix tree

Examples with Explanations

Delete a group

```
getent group oldteam
sudo groupdel oldteam
getent group oldteam || echo gone
```

Refuse if primary group

```
sudo groupdel alice
# groupdel: cannot remove the primary group of user 'alice'
```

Change users' primary group first:

```
sudo usermod -g users alice
sudo groupdel alice
```

Check membership first

```
getent group devs
# see members field
grep -E '^devs:' /etc/group
sudo libuser-groupdel 2>/dev/null # alternate tools exist
# list users with primary gid:
gid=$(getent group devs | cut -d: -f3)
awk -F: -v g="$gid" '$4==g {print $1}' /etc/passwd
```

Automation cleanup

```
if getent group tempci >/dev/null; then
    sudo groupdel tempci
fi
```

LDAP / remote groups

```
getent group eng
# if from sssd/ldap, groupdel on local files won't remove directory groups
```

Use your directory admin tools for networked accounts.

Notes / Pitfalls

- Files previously group-owned keep the **numeric GID** — they become “orphan” GIDs in `ls -l` until reowned.
- Does not modify ACLs that referenced the group name (may become unresolved).
- Always inventory with `getent group` and filesystem owners before deleting shared groups.
- Busy production groups (e.g. shared data) need migration of ownership first.
- Containers: deleting groups inside a container doesn't affect the host.

2026-relevant notes

- Prefer configuration management (Ansible users modules) over one-off groupdel on fleets.
- System groups for systemd DynamicUser units are managed differently — don't manually delete unknown system groups.
- Audit file ownership: `find / -group oldgid` can be expensive; scope paths.

Related Commands

- `groupadd / groupmod` — create / modify
- `useradd / usermod` — user membership
- `getent group` — lookup
- `gpasswd` — administer group members
- `chgrp / chown` — fix file ownership

Additional Resources

- `man groupdel`

getent

Overview

`getent` queries **Name Service Switch (NSS)** databases—the same view applications get via `glibc`: local files, LDAP/SSSD, mDNS, DNS for hosts, and more. Prefer it over `cat /etc/passwd` when directory services or `nsswitch` plugins are in play.

Syntax

```
getent [options] database [key ...]
```

Common Databases

Database	Purpose
<code>passwd</code>	Users
<code>group</code>	Groups
<code>shadow</code>	Shadow passwords (privileged)
<code>hosts</code>	Host lookup
<code>ahosts</code> / <code>ahostsv4</code> / <code>ahostsv6</code>	<code>getaddrinfo</code> -style
<code>services</code>	Service name port
<code>protocols</code>	Protocol names
<code>networks</code>	Network names
<code>aliases</code>	Mail aliases (if used)
<code>ethers</code>	Hardware addresses (rare)

Common Options

Option	Description
<code>-s SERVICE</code>	Force NSS service
<code>-i</code>	Case insensitive where supported

Key Use Cases

1. Confirm a user exists in LDAP/SSSD, not only locally
2. Script portable account lookups
3. Debug host resolution order (files/dns)
4. Map service names to ports

Examples with Explanations

Users and groups

```
getent passwd alice
getent passwd 1000
getent group sudo
getent group 27
```

Hosts

```
getent hosts example.com
getent ahosts example.com
getent hosts 1.1.1.1
```

Services

```
getent services ssh
getent services 443/tcp
```

One-liner recipes

```
# Does this UID resolve?
getent passwd 1001 || echo 'unknown uid'

# Members visible via NSS
getent group developers

# Compare file vs NSS
getent passwd alice; grep '^alice:' /etc/passwd || true

# nsswitch insight
cat /etc/nsswitch.conf
```


Notes & Pitfalls

- Empty result means “not found in any configured source,” not necessarily “query error.”
- `shadow` access is restricted; applications should use proper auth APIs.
- Host lookups follow `nsswitch.conf`—may never touch DNS if `files` hits first.
- Enumerating all LDAP users with bare `getent passwd` can be huge or disabled—don’t DoS your directory.

2026-relevant notes

- SSSD/FreeIPA/Azure AD-ish integrations still show up through NSS; `getent` is the smoke test.
- For DNS-only questions use `dig/resolvectl`; `getent hosts` includes non-DNS sources.
- Containers often have minimal `passwd`; UIDs from the host may not resolve by name inside.

Related Commands

- `id` — UIDs/GIDs for a user
- `passwd` / `usermod` — local account changes
- `dig` / `resolvectl` — pure DNS
- `getent` vs `cat /etc/passwd` — NSS is authoritative for apps

Additional Resources

- `man getent`, `man nsswitch.conf`

SU

Overview

`su` (“substitute user”) starts a shell or runs a command as another user. On modern Ubuntu, prefer `sudo` for policy-controlled elevation; `su` still matters for switching to service accounts, recovering systems without `sudo`, and older procedures. `runuser` is a non-setuid alternative used in scripts from root.

Syntax

```
su [options] [-] [user [argument...]]
su [options] -c command [user]
```

Common Options

Option	Description
<code>- / -l / --login</code>	Login shell (loads target profile, resets env)
<code>-c cmd</code>	Run command then exit
<code>-s shell</code>	Use alternate shell
<code>-w / --whitelist-environment</code>	Preserve listed vars (util-linux)
<code>-p / -m</code>	Preserve environment (careful)

Safety

- `su -` vs `su` differs wildly in environment (`HOME`, `PATH`). Prefer login form for interactive admin.
- From an unprivileged user, `su` asks for the **target** user’s password (root password on many systems) — not your `sudo` password.
- Avoid `su -c` with unquoted complex scripts; quoting bugs become root bugs.

Examples with Explanations

Become root (login)

```
su -  
# or  
su - root
```

Run one command as root

```
su -c 'systemctl status ssh' root
```

Switch to a service user

```
sudo su - postgres  
sudo -u postgres -i          # usually better when sudo is available
```

Preserve selected environment

```
su -w LANG,TERM - alice
```

Notes & Pitfalls

- Many Ubuntu systems (including Server) have **no root password** by default — `su` to root fails; use `sudo -i`.
- PAM modules may restrict `su` even if the password is known.
- `sudo su` works but obscures audit trails compared to `sudo -i` / direct `sudo cmd`.

Related Commands

- `sudo` — policy-based elevation
- `runuser` — switch user from root without `setuid su`
- `login` — full login path
- `machinectl shell` — containers/machines

Additional Resources

- `man su`
- `man runuser`

Networking

Intro

Intro

Configure and diagnose connectivity: addresses and routes (**ip**), sockets (**ss**), DNS, HTTP clients, SSH/file transfer, firewalls, and packet capture. Prefer modern tools (**ip/ss/nft**) over legacy ones.

Commands in this part

Command	Role
ip	ip (from iproute2) configures and displays interfaces, addresses, routes, neighbors, rules, and tunnels.
ss	ss (socket statistics) inspects TCP/UDP/UNIX sockets.
ping	ping sends ICMP Echo Request packets and reports whether a host answers with Echo Reply, plus round-trip time (RTT).
traceroute	traceroute maps the path packets take toward a destination by sending probes with increasing TTL and reading ICMP...
mtr	mtr (My Traceroute) combines continuous ping and traceroute: it probes each hop on the path and updates loss and...
dig	dig (Domain Information Groper) queries DNS servers and prints detailed answers.
host	host is a simple DNS lookup utility from BIND tools.
nslookup	nslookup queries DNS name servers interactively or non-interactively.
doggo	doggo is a modern command-line DNS client with colorful, human-friendly output.

Command	Role
<code>resolvectl</code>	<code>resolvectl</code> manages and queries <code>systemd-resolved</code> : DNS servers, search/route domains, LLMNR/mDNS, DNSSEC/DoT status,...
<code>curl</code>	<code>curl</code> transfers data to/from URLs.
<code>wget</code>	<code>wget</code> is a non-interactive downloader for HTTP(S), FTP, and related protocols.
<code>ssh</code>	<code>ssh</code> (OpenSSH client) creates encrypted sessions to a remote host for shell access, command execution, port...
<code>scp</code>	<code>scp</code> (secure copy) copies files to or from a remote host over SSH.
<code>rsync</code>	<code>rsync</code> synchronizes files and directories locally or over a remote shell (almost always SSH).
<code>nc</code>	<code>nc</code> (netcat) is a Swiss-army TCP/UDP tool: open connections, listen for inbound sockets, scan ports simply, and pipe...
<code>tcpdump</code>	<code>tcpdump</code> captures and displays packets on a network interface.
<code>nmap</code>	The <code>nmap</code> (Network Mapper) is a powerful open-source tool for network exploration, security scanning, and auditing.
<code>openssl</code>	<code>openssl</code> is a Swiss-army toolkit for TLS/SSL, X.509 certificates, keys, digests, and simple encrypted files.
<code>nmcli</code>	<code>nmcli</code> is the CLI for NetworkManager.
<code>ufw</code>	<code>ufw</code> (Uncomplicated Firewall) is Ubuntu's friendly frontend to <code>netfilter</code> .
<code>nft</code>	<code>nft</code> is the userspace CLI for <code>nftables</code> , the modern Linux packet-filtering framework that replaces legacy <code>iptables</code> ...
<code>iptables</code>	<code>iptables</code> configures IPv4 packet filtering/NAT via <code>netfilter</code> .
<code>ifconfig</code>	<code>ifconfig</code> configures and displays network interfaces from the legacy <code>net-tools</code> package.
<code>netstat</code>	<code>netstat</code> displays network connections, listening sockets, routing tables, and interface statistics from the legacy...

Command	Role
<code>ftp</code>	<code>ftp</code> is the classic interactive client for the File Transfer Protocol.
<code>telnet</code>	<code>telnet</code> is a historic remote terminal protocol client.
<code>ssh-keygen</code>	<code>ssh-keygen</code> creates and manages SSH key pairs, fingerprints host keys, and can update <code>known_hosts</code> .
<code>ssh-copy-id</code>	<code>ssh-copy-id</code> installs your public key on a remote account's <code>~/.ssh/authorized_keys</code> with safer permissions than...
<code>sftp</code>	<code>sftp</code> is the interactive SSH file-transfer client (SFTP subsystem).
<code>ethtool</code>	<code>ethtool</code> queries and controls Ethernet NIC settings: link speed/duplex, offloads, ring parameters, and driver info.
<code>firewall-cmd</code>	<code>firewall-cmd</code> is the client for <code>firewalld</code> , the dynamic firewall manager on Fedora, RHEL, CentOS Stream, and many...

Suggested starting points

1. Local stack: `ip`, `ss`, `nmcli/resolvectl` as appropriate.
2. Path checks: `ping`, `traceroute/mtr`, then `dig/host` for DNS.
3. HTTP and APIs: `curl` (and `wget` for simple downloads).
4. Remote admin: `ssh`, keys via `ssh-keygen/ssh-copy-id`, files via `scp/sftp/rsync`.
5. Firewalls: `ufw` (Ubuntu), `firewall-cmd` (`firewalld`), or `nft/iptables`.
6. Deep debug: `tcpdump`, `nmap`, `openssl`, `ethtool`.

Related parts

- System monitoring — correlate with host load
- Security — host hardening beyond the firewall
- Services and runtime — network targets and sockets

Continue with the individual command pages in this part.

ip

Overview

`ip` (from **iproute2**) configures and displays interfaces, addresses, routes, neighbors, rules, and tunnels. It replaces legacy `ifconfig/route/arp` for modern Linux networking.

Syntax

```
ip [OPTIONS] OBJECT COMMAND
ip -h                # help
ip OBJECT help       # object-specific help
```

Common objects: `link`, `address (addr)`, `route`, `neigh`, `rule`, `netns`, `xfrm`.

Common Options

Option	Description
<code>-4 / -6</code>	IPv4 / IPv6 only
<code>-s</code>	Statistics (can repeat for more detail)
<code>-d</code>	Extra detail
<code>-br</code>	Brief columnar output
<code>-c</code>	Color
<code>-o</code>	One-line output
<code>-j / -p</code>	JSON / pretty JSON (newer iproute2)

Key Use Cases

1. Inspect and set addresses
2. Bring links up/down, set MTU/MAC
3. Manage default and static routes

4. Debug neighbor (ARP/ND) tables
5. Work with network namespaces

Safety

Address and route changes can drop your SSH session. Prefer a **console/IPMI** session or a scripted rollback when changing the only default route. On Ubuntu servers, prefer **netplan** or **NetworkManager** for persistent config; use **ip** for live debug.

Examples with Explanations

Brief overview

```
ip -br link
ip -br addr
ip route
```

Fast “what interfaces, what IPs, what default route?”.

Full address list

```
ip addr show
ip -4 addr show dev eth0
```

Bring link up/down

```
sudo ip link set dev eth0 up
sudo ip link set dev eth0 down
```

Add/remove an address

```
sudo ip addr add 192.168.1.50/24 dev eth0
sudo ip addr del 192.168.1.50/24 dev eth0
```

Prefix length is required. Secondary addresses are normal on multi-homed hosts.

Default route

```
sudo ip route add default via 192.168.1.1 dev eth0
sudo ip route replace default via 192.168.1.1 dev eth0
ip route get 1.1.1.1
```

`route get` shows which path the kernel would use.

Static route

```
sudo ip route add 10.0.0.0/8 via 192.168.1.254
sudo ip route del 10.0.0.0/8
```

Neighbor table (ARP)

```
ip neigh show
sudo ip neigh flush dev eth0
```

Statistics

```
ip -s link show eth0
ip -s -s link show eth0  # more error detail
```

Network namespace (quick)

```
sudo ip netns add lab
sudo ip netns exec lab ip link
sudo ip netns delete lab
```

Understanding Output

- UP/LOWER_UP on links: admin up vs carrier present
- inet / inet6 lines: addresses and scopes (global, link)
- proto kernel vs proto static / dhcp on routes: origin of the route

Common Usage Patterns

Live IP on a server you did not configure

```
ip -br a; ip r; resolvectl status 2>/dev/null || cat /etc/resolv.conf
```

Flush all IPv4 addresses on a NIC (careful)

```
sudo ip -4 addr flush dev eth0
```

Notes & Pitfalls

- Changes with `ip` are **not durable** across reboot unless persisted by netplan/NM/scripts.
- Interface names may be predictable (`enp3s0`) not `eth0`.
- Policy routing uses `ip rule` + multiple tables — advanced; see `man ip-rule`.

Related Commands

- `nmcli` — NetworkManager CLI
- `resolvectl` — DNS via systemd-resolved
- `ss` — sockets
- `ping` / `traceroute` — path checks
- `ifconfig` — legacy; prefer `ip`

Additional Resources

- `man ip`, `man ip-link`, `man ip-route`
- `iproute2` documentation

SS

Overview

ss (socket statistics) inspects TCP/UDP/UNIX sockets. It replaces most **netstat** workflows and is faster on busy systems because it talks to the kernel more efficiently. First stop for “what is listening?” and “who is connected?”.

Syntax

```
ss [options] [filter]
```

Common Options

Option	Description
-t / -u / -x	TCP / UDP / UNIX
-l	Listening only
-a	All (listen + connected)
-n	Numeric hosts/ports (no DNS)
-p	Owning process (often needs root)
-e	Extended detail
-m	Socket memory info
-s	Summary counters
-4 / -6	Address family
-o	Timer info (retransmit, keepalive)
-K	Kill socket (privileged; careful)
-H	Suppress header (newer ss)

Key Use Cases

1. See listeners and bound ports
2. Map sockets to PIDs/processes

3. Find connections to/from a host or port
4. Spot TIME-WAIT piles and backlog issues

Examples with Explanations

Listening TCP with processes

```
sudo ss -lntp
```

-n avoids reverse DNS delays; -p shows users:(("nginx",pid=...)).

Established connections

```
ss -tnp
ss -tnp state established
```

Filter by port

```
sudo ss -lntp '( sport = :22 )'
sudo ss -lntp '( sport = :8080 )'
ss -tnp '( dport = :443 or sport = :443 )'
```

Quote filters so the shell does not eat parentheses.

Filter by address

```
ss -tn dst 1.1.1.1
ss -tn src 10.0.0.0/8
```

UDP listeners

```
sudo ss -lunp
```

DNS, NTP, QUIC-related services often show here.

Summary and states

```
ss -s
ss -ant | awk 'NR>1 {c[$1]++} END {for (s in c) print c[s], s}' | sort -nr
```

High TIME-WAIT can be normal after load; runaway SYN-RECV may mean flood or backlog misconfig.

UNIX domain sockets

```
sudo ss -xlp | head
ss -x src /run/systemd/private
```

Local IPC for docker/podman/systemd often appears as UNIX sockets.

“What owns this port?”

```
sudo ss -lntp | grep -E ':8080\b'
# or filter expression:
sudo ss -lntp '( sport = :8080 )'
sudo lsof -iTCP:8080 -sTCP:LISTEN
```

IPv6 only

```
sudo ss -lntp -6
```

Understanding Output

Typical columns: state (LISTEN, ESTAB, TIME-WAIT, ...), receive/send queues, local address:port, peer address:port, process. Queue columns matter when diagnosing overload (apps not `accept()`ing fast enough).

Notes & Pitfalls

- Without `sudo`, `-p` may omit other users’ processes.
- `netstat` still appears in old docs; install `iproute2` (`ss`) as the default.
- Filters use a small language — see `man ss` “FILTER”.
- For payload inspection use `tcpdump`/`wireshark`, not `ss`.

Related Commands

- `lsof -i` — open-file oriented view
- `ip` — addresses and routes
- `nmap` — external reachability scans
- `netstat` — legacy
- `fuser` — who uses a port/file

Additional Resources

- `man ss`

ping

Overview

`ping` sends ICMP Echo Request packets and reports whether a host answers with Echo Reply, plus round-trip time (RTT). It is the first tool for “is this address reachable?” and a quick signal for loss and latency. On Ubuntu, `ping` comes from `iputils-ping` and needs appropriate capabilities (usually installed `setuid` or with `cap_net_raw`) to open raw ICMP sockets.

Syntax

```
ping [OPTIONS] DESTINATION
ping -c COUNT [OPTIONS] DESTINATION
```

Common Options

Option	Description
<code>-c count</code>	Stop after <code>count</code> replies (script-friendly)
<code>-i interval</code>	Seconds between packets (default 1)
<code>-W timeout</code>	Seconds to wait for a reply
<code>-w deadline</code>	Exit after this many seconds regardless
<code>-s size</code>	Payload size in bytes
<code>-M do</code>	Path MTU discovery: prohibit fragmentation
<code>-4 / -6</code>	Force IPv4 / IPv6
<code>-n</code>	Numeric output (skip reverse DNS)
<code>-q</code>	Quiet (summary only)
<code>-v</code>	Verbose
<code>-I device</code>	Bind to interface or source address
<code>-t ttl</code>	Set IP TTL

Safety

- Flooding (`-f`, if permitted) or very low intervals can look like abuse and may be rate-limited by networks. Prefer `-c` and sensible `-i` on production paths.

- “Ping works” “application works.” ICMP may be filtered while TCP 443 is fine, and the reverse can also happen.
- Do not rely on ping alone for SLOs; use service health checks.

Examples with Explanations

Basic connectivity (4 probes)

```
ping -c 4 1.1.1.1
ping -c 4 example.com
```

Numeric only (faster, no rDNS noise)

```
ping -n -c 4 8.8.8.8
```

IPv6

```
ping -6 -c 4 2606:4700:4700::1111
# or:
ping6 -c 4 example.com
```

Quiet summary for scripts

```
if ping -c 3 -W 2 -q gateway.local >/dev/null; then
    echo reachable
else
    echo unreachable
fi
```

Deadline: fail fast in automation

```
ping -c 5 -w 6 -q 203.0.113.1
echo exit:$?
```

`-w` caps total runtime so a black hole does not hang a script.

Larger payload (rough MTU / fragmentation check)

```
ping -c 3 -s 1472 -M do 1.1.1.1
```

1472 payload + 28 bytes IP/ICMP headers = 1500-byte Ethernet frame. If this fails with “message too long” but small pings work, look at MTU/PMTUD issues (VPNs, tunnels).

Source from a specific interface

```
ping -c 4 -I eth0 10.0.0.1
ping -c 4 -I 192.0.2.10 10.0.0.1
```

Useful on multi-homed hosts and VPN boxes.

Faster sampling (still polite)

```
ping -c 20 -i 0.2 10.0.0.1
```

Requires privileges for intervals < 0.2s on many systems; stay reasonable on shared networks.

Compare loss to a hop vs destination

```
ping -c 50 -q 10.0.0.1    # gateway
ping -c 50 -q 1.1.1.1    # internet
```

High loss only to the far end suggests a path problem beyond the LAN.

Understanding Output

Typical line:

```
64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=12.3 ms
```

- **icmp_seq** — sequence number; gaps imply loss (or reordering).
- **ttl** — remaining hop budget; lower often means more hops (not a strict hop count).
- **time** — RTT.

Summary:

```
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 11.2/12.5/14.1/1.1 ms
```

- **mdev** — mean deviation (jitter-ish signal).
- Exit status: 0 if enough replies received (see man page for exact rules with `-c/-w`), non-zero on total failure.

Notes & Pitfalls

- Many cloud security groups and enterprise firewalls **drop ICMP**. Unreachable ping does not prove the host is down — try `curl -I`, `nc -vz host 443`, or `mtr`.
- Name resolution failures look like ping failures; test with a raw IP too.
- Container and network-namespace environments may lack permission to ping; error messages about “Operation not permitted” point at capabilities, not routing.
- Do not confuse **ping** host reachability with DNS: always note whether you used a name or an IP.
- For continuous path quality, `mtr` or `smokeping`-style tools beat a single short ping.

Related Commands

- `mtr / mtr --report` — live traceroute + loss per hop
- `traceroute / tracepath` — path discovery
- `ss / ip route` — local sockets and routes
- `curl -I / nc -vz` — application-layer reachability
- `resolvectl query / dig` — name resolution checks

Additional Resources

- `man ping`
- `man ip-route` (when replies fail but the host seems local)

traceroute

Overview

`traceroute` maps the path packets take toward a destination by sending probes with increasing TTL and reading ICMP “Time Exceeded” messages from intermediate hops. Use it to find where latency jumps or where the path dies. On Ubuntu, install with `sudo apt install traceroute`. Related tools: `tracepath` (often installed by default), and `mtr` (continuous combined ping+trace).

Syntax

```
traceroute [options] host
traceroute -n -m 20 host
```

Common Options

Option	Description
<code>-n</code>	Numeric IPs only (no reverse DNS — faster, cleaner)
<code>-m max_ttl</code>	Max hops (default often 30)
<code>-q nqueries</code>	Probes per hop (default 3)
<code>-w wait</code>	Wait seconds per probe
<code>-I</code>	ICMP echo probes (may work when UDP is filtered)
<code>-T</code>	TCP SYN probes (often best through firewalls)
<code>-U</code>	UDP probes (classic default on many Linux builds)
<code>-p port</code>	Base port (UDP) or destination port (TCP with <code>-T</code>)
<code>-f first_ttl</code>	Start at this TTL
<code>-4 / -6</code>	IPv4 / IPv6
<code>-i iface</code>	Outgoing interface
<code>-s src</code>	Source address

Safety

- Traceroute is generally benign but generates many probes; do not hammer production paths in tight loops.

- Some providers rate-limit or deprioritize traceroute-related ICMP; interpret stars (* * *) carefully.
- TCP traceroute to sensitive ports may look like a scan — prefer known service ports you own (e.g. 443) and authorized targets.

Examples with Explanations

Basic path

```
traceroute example.com
traceroute -n 1.1.1.1
```

ICMP mode (when UDP probes are blocked)

```
sudo traceroute -I -n example.com
```

May require root for raw ICMP.

TCP traceroute to HTTPS (firewall-friendly)

```
sudo traceroute -T -p 443 -n example.com
```

Probes look more like real client traffic to port 443; excellent for “UDP traceroute dies but the site loads” cases.

Limit hop count and speed up failure

```
traceroute -n -m 15 -w 2 -q 1 203.0.113.50
```

-q 1 one probe per hop; -w 2 shorter wait — noisier stats but faster.

IPv6 path

```
traceroute -6 -n example.com
```

Compare UDP vs TCP vs ICMP

```
tracert -n example.com
sudo tracert -I -n example.com
sudo tracert -T -p 443 -n example.com
```

Different probe types can take different paths or be filtered differently — note which method matches user traffic.

Start mid-path (skip nearby hops)

```
tracert -n -f 5 example.com
```

Use a specific source address (multi-homed)

```
sudo tracert -n -s 192.0.2.10 example.com
```

Understanding Output

Classic lines:

```
1  192.168.1.1  1.2 ms  1.1 ms  1.3 ms
2  10.0.0.1     5.0 ms  4.8 ms  5.1 ms
3  * * *
4  203.0.113.1 20.2 ms 19.8 ms 20.5 ms
```

- Column 1 — hop number (TTL).
- Address — router that returned ICMP time exceeded (or final destination).
- Three times — RTTs for each probe (-q).
- * * * — no reply within timeout (filtering, loss, or rate-limit) — **not always a failure of the path.**

Notes on interpretation: - Latency often rises gradually; a sudden jump may mark a long-haul or congested link. - Asymmetric routing means the return path may differ; RTT includes both directions. - The last hop should be the destination answering the probe type; some hosts never respond to traceroute probes yet still serve TCP apps.

Notes & Pitfalls

- **Stars in the middle do not prove breakage** if later hops respond. Many routers simply do not generate TTL-exceeded for your probe type.

- Stars at the **end** with a working website often mean the destination ignores traceroute probes; test with `curl/nc -vz` too.
- Corporate firewalls may allow TCP/443 but drop UDP traceroute — use `-T -p 443`.
- Reverse DNS (`-n off`) slows output and can mislead if PTR records are wrong; prefer `-n` for ops work.
- `tracepath` is a lighter alternative that does not always need root and focuses on PMTU; `mtr --report` is better for ongoing loss per hop.
- ECMP / anycast can make successive runs show different hop addresses — capture multiple runs before concluding.

Related Commands

- `mtr / mtr --report` — continuous per-hop loss and latency
- `tracepath` — simple path / PMTU discovery
- `ping` — end-to-end reachability and RTT
- `ip route get ADDR` — which local route/next-hop would be used
- `dig + curl` — confirm name and application path independently

Additional Resources

- `man traceroute`
- `man mtr`
- `man tracepath`

mtr

Overview

mtr (My Traceroute) combines continuous **ping** and **traceroute**: it probes each hop on the path and updates loss and latency statistics in real time. Use it when intermittent packet loss or a “bad hop” is suspected—not just for a one-shot path dump.

```
# Debian/Ubuntu / RHEL-family (package name is usually mtr or mtr-tiny)
sudo apt install mtr-tiny    # or: mtr
sudo dnf install mtr
```

Syntax

```
mtr [options] HOSTNAME|IP
```

Common Options

Option	Description
-r, --report	Non-interactive report then exit
-c N, --report-cycles N	Probes per hop in report mode
-n, --no-dns	Skip reverse DNS (faster, clearer)
-w, --report-wide	Wide hostnames in reports
-4 / -6	Force IPv4 or IPv6
-T, --tcp	TCP SYN probes instead of ICMP
-u, --udp	UDP probes
-P PORT, --port PORT	Target port for TCP/UDP
-i SECS	Interval between probes
-m MAX	Max hops (TTL)
-b	Show both hostname and IP
-z	Show AS numbers (if available)
-s SIZE	Packet size
-e	Show MPLS labels when present

Key Use Cases

1. Locate which hop introduces latency or loss
2. Generate shareable path reports for NOC tickets
3. Test path to a specific TCP port (HTTPS 443, etc.)
4. Compare IPv4 vs IPv6 paths

Examples with Explanations

Interactive session

```
mtr example.com  
mtr -n example.com
```

Interactive UI refreshes hop stats until you quit (q). `-n` avoids slow or broken PTR lookups.

Non-interactive report (scripts & tickets)

```
mtr --report -c 20 -n example.com  
mtr -rwc 50 example.com
```

`--report` runs N cycles and prints a table—ideal for pasting into chat or logs.

TCP path to HTTPS

```
mtr -T -P 443 -n example.com  
mtr -T -P 22 -n bastion.example.com
```

ICMP is often rate-limited or deprioritized; TCP probes better match real application traffic when firewalls allow them.

IPv6 and dual-stack

```
mtr -6 -n example.com  
mtr -4 -n example.com
```

Different address families can take completely different transit.

One-liner recipes

```
# Quick 10-cycle report to clipboard-friendly text
mtr -rwc 10 -n 1.1.1.1

# Compare two destinations
for h in 1.1.1.1 8.8.8.8; do echo "=== $h ==="; mtr -rwc 15 -n "$h"; done

# Watch path while changing VPN/Wi-Fi (interactive)
mtr -n vpn-endpoint.example.com
```

Interpreting loss (important)

- **Loss only on intermediate hops** but 0% at the destination often means the hop rate-limits ICMP replies—not necessarily real path loss.
- **Loss that grows and stays high through later hops including destination** usually indicates a real problem on or before that hop.
- Spike latency on one hop that recovers downstream can be ICMP deprioritization.

Understanding Output

Typical columns:

Field	Meaning
Host	Hop address / name
Loss%	Probe loss to that hop
Snt	Probes sent
Last / Avg / Best / Wrst	RTT samples (ms)
StDev	Jitter-ish spread

Notes & Pitfalls

- **Root/capabilities:** some probe types need elevated privileges; distro packages may install a setcap binary.
- **Firewalls:** corporate networks may drop traceroute-style probes; try `-T -P 443`.
- **Containers / cloud:** path may terminate at hypervisor or LB; don't expect full Internet-style hops inside VPC only.
- Prefer **mtr** over **pure traceroute** + **ping** for intermittent issues.

2026-relevant notes

- Cloud anycast and CGNAT make hop lists noisier; combine with `curl -w '%{time_connect}\n'`, `ss`, and provider path tools.
- For continuous SLOs, export `mtr --report` from a cron/systemd timer rather than relying on ad-hoc interactive sessions.
- `mtr-tiny` on Debian omits some extras but is fine for servers.

Comparison to alternatives

Tool	Role
<code>mtr</code>	Continuous hop stats
<code>tracert</code> / <code>tracert</code>	One-shot path
<code>ping</code>	Single-host RTT/loss
<code>nping</code> / <code>smokeping</code>	Long-term historical

Related Commands

- `ping` — single-host reachability
- `tracert` / `tracert` — path discovery
- `ss` — local sockets
- `dig` / `host` — name resolution before tracing

Additional Resources

- `man mtr`
- Path diagnosis guides from major cloud and CDN providers

dig

Overview

dig (Domain Information Groper) queries DNS servers and prints detailed answers. It is the preferred CLI for debugging resolution on Ubuntu/Debian (package **dnsutils** / **bind9-dnsutils**). Use it to inspect A/AAAA/MX/TXT/NS/CNAME records, compare resolvers, and validate changes after DNS edits.

Syntax

```
dig [@server] [name] [type] [options]
dig [options] name type
```

Default type is A. If **@server** is omitted, **dig** uses the system resolver configuration (often **systemd-resolved** stub on Ubuntu).

Common Options

Option	Description
@server	Query this nameserver (IP or hostname)
-t type / positional type	Record type: A, AAAA, MX, TXT, NS, CNAME, SOA, ANY (limited), ...
+short	Minimal answer only
+noall +answer	Show answer section only (structured)
+trace	Iterative resolve from root hints
+dnssec	Request DNSSEC records
+tcp	Use TCP instead of UDP
+time=T	Timeout seconds
+tries=N	Number of tries
-p port	Nameserver port (default 53)
-x addr	Reverse lookup (PTR) convenience
-4 / -6	Transport IPv4 / IPv6 only
+subnet=...	EDNS client subnet (advanced)

Examples with Explanations

Basic A record

```
dig example.com
dig example.com A
```

Short output for scripts

```
dig +short example.com
dig +short example.com AAAA
```

Query a specific resolver

```
dig @1.1.1.1 example.com
dig @8.8.8.8 example.com AAAA
dig @127.0.0.53 example.com # systemd-resolved stub on many Ubuntu hosts
```

Comparing @1.1.1.1 vs system resolver isolates “DNS server cache/policy” vs “local stub” issues.

MX and TXT

```
dig example.com MX +short
dig example.com TXT +short
```

NS and SOA (delegation / zone authority)

```
dig example.com NS +short
dig example.com SOA
```

Reverse DNS (PTR)

```
dig -x 1.1.1.1 +short
# equivalent idea:
dig 1.1.1.1.in-addr.arpa PTR +short
```

Answer section only (readable)

```
dig +noall +answer example.com A
dig +noall +answer www.example.com
```

Trace from the root (find where delegation breaks)

```
dig +trace example.com
```

Shows root → TLD → authoritative path. Heavy but excellent when “some resolvers see X, others Y.”

Follow CNAME manually

```
dig www.example.com +short
dig +noall +answer www.example.com
```

If you get a CNAME, dig again on the target name for the final A/AAAA.

TCP query (firewall / large answers)

```
dig +tcp example.com TXT
```

Non-standard port (lab DNS)

```
dig @127.0.0.1 -p 5353 test.lab A
```

Batch mode

```
dig -f names.txt +short
```

names.txt lists one name per line.

Check mail-related records quickly

```
dig example.com MX +short
dig example.com TXT +short | grep -i spf
dig _dmarc.example.com TXT +short
```


Understanding Output

Default output sections: - **HEADER** — opcode, status (NOERROR, NXDOMAIN, SERVFAIL, REFUSED), flags (**aa** authoritative, **rd** recursion desired, **ra** recursion available). - **QUESTION** — what you asked. - **ANSWER** — records for the name. - **AUTHORITY** — NS/SOA when relevant. - **ADDITIONAL** — glue / extras. - **Query time / SERVER / WHEN / MSG SIZE** — which server answered and how fast.

Status guide: | Status | Meaning | |———|———| | NOERROR | Query ok (answer may still be empty for that type) | | NXDOMAIN | Name does not exist | | SERVFAIL | Resolver failure (upstream, DNSSEC, timeout) | | REFUSED | Server refused the query |

Notes & Pitfalls

- Ubuntu's 127.0.0.53 is often a **stub** — it forwards to real resolvers. Debugging “wrong answer” may need @ the upstream (`resolvectl status`) or a public resolver.
- ANY is deprecated/restricted on many public resolvers; query specific types instead.
- TTL in answers is **remaining** TTL from that resolver's cache, not necessarily the zone's configured TTL.
- Split-horizon DNS returns different answers by client network; VPN on/off can change results.
- `dig` success does not mean your application uses the same resolver path — check `resolvectl`, `/etc/nsswitch.conf`, and app-specific DNS settings.
- For quick human checks, `host` is shorter; for operators, `dig` is the right default.

Related Commands

- `host` — shorter DNS lookups
- `nslookup` — interactive/legacy style lookups
- `resolvectl query` — systemd-resolved view on Ubuntu
- `doggo` / `drill` — alternative dig-like tools if installed
- `whois` — registration data (not live DNS answers)

Additional Resources

- `man dig`
- `man resolvectl` (Ubuntu resolver integration)

host

Overview

`host` is a simple DNS lookup utility from BIND tools. It is faster to type than `dig` for everyday A/AAAA/MX checks. Use `dig` (or `drill`) when you need full message sections, DNSSEC detail, or uncommon query flags; use `resolvectl query` when you care what *this machine's* stub resolver returns.

Syntax

```
host [options] name [server]
host [options] IP
```

Common Options

Option	Description
<code>-t type</code>	Record type: A, AAAA, MX, TXT, NS, CNAME, SOA, SRV, ...
<code>-a</code>	Equivalent to <code>-v -t ANY</code> (ANY often blocked)
<code>-v</code>	Verbose
<code>-W secs</code>	Wait timeout
<code>-R n</code>	Number of retries
<code>-C</code>	Query SOA records for zone name server check style
<code>-4 / -6</code>	Force IPv4/IPv6 transport to the nameserver
<code>-r</code>	Non-recursive query

Key Use Cases

1. Quick “does this name resolve?”
2. MX / TXT / NS checks without dig verbosity
3. Reverse lookup of an IP
4. Point queries at a specific nameserver

Examples with Explanations

Forward lookup

```
host example.com
host -t A example.com
host -t AAAA example.com
```

Default output lists addresses and sometimes other associated data depending on version and response.

Mail and text records

```
host -t MX example.com
host -t TXT example.com
host -t NS example.com
host -t SOA example.com
```

Reverse DNS

```
host 1.1.1.1
host 8.8.8.8
```

PTR for the reverse zone; empty/no PTR is common for ephemeral cloud IPs.

Query a specific server

```
host example.com 1.1.1.1
host example.com 8.8.8.8
host example.com 127.0.0.53
```

Compare public DNS vs local systemd-resolved stub.

Verbose and script-friendly

```
host -v example.com
if host -W 2 example.com >/dev/null 2>&1; then echo ok; else echo fail; fi
```

One-liner recipes

```
# Extract first IPv4-looking answer (fragile but handy)
host -t A example.com | awk '/has address/ {print $4; exit}'

# MX hosts only
host -t MX example.com | awk '{print $NF}'

# Spot split-horizon DNS
diff <(host example.com 1.1.1.1) <(host example.com 127.0.0.53) || true
```

Notes & Pitfalls

- **ANY** (-a) is widely restricted on public resolvers—query explicit types.
- Output format is **not as stable as dig + short** for parsing; prefer **dig +short** or **doggo --json** in automation.
- **Search domains** from resolv.conf can make short names resolve differently than FQDNs.
- **host** may not be installed on minimal images (**bind-utils** / **dnsutils** / **bind9-host** packages).

2026-relevant notes

- Prefer **dig +short**, **doggo**, or **resolvectl query** for modern workflows; keep **host** for muscle memory and quick checks.
- Encrypted DNS is usually configured under resolved/NetworkManager—not via **host** flags. To test DoH, use **doggo** or a DoH-aware client.
- **drill** (ldns) is a good dig-like alternative on some distros.

Comparison to alternatives

Tool	When to use
host	Short interactive answers
dig	Full protocol control
doggo	UX + DoH/DoT
nslookup	Legacy interactive only
getent hosts	NSS (files + DNS + others)

Related Commands

- `dig` — detailed DNS
- `nslookup` — legacy
- `resolvectl` — `systemd-resolved`
- `doggo` — modern CLI
- `getent hosts` — NSS

Additional Resources

- `man host`
- BIND documentation for the `host` utility

nslookup

Overview

nslookup queries DNS name servers interactively or non-interactively. It remains common in legacy docs and Windows cross-training, but on modern Linux **dig**, **host**, **doggo**, and **resolvel** **query** are usually better choices for clarity and scripting.

Syntax

```
nslookup [options] [name | -] [server]
```

Common Options / Directives

Option / interactive	Description
-type=a / type=a	A record
-type=aaaa	AAAA
-type=mx	MX
-type=ns	NS
-type=soa	SOA
-type=txt	TXT
-type=ptr	PTR
-type=cname	CNAME
-debug	Debug
-port=N	Server port
-timeout=N	Timeout
server ADDR	Interactive: change server
set type=mx	Interactive type

Key Use Cases

1. Quick interactive DNS checks
2. Following older runbooks
3. Comparing answers from a specific server

Examples with Explanations

Non-interactive

```
nslookup example.com
nslookup -type=mx example.com
nslookup -type=ns example.com
nslookup example.com 1.1.1.1
nslookup 1.1.1.1
```

Interactive

```
nslookup
> server 8.8.8.8
> set type=txt
> example.com
> exit
```

Prefer modern equivalents

```
dig +short example.com
dig example.com MX
host -t MX example.com
resolvectl query example.com
doggo example.com MX
```

One-liner recipes

```
# Script-friendly? Prefer dig:
dig +short example.com @1.1.1.1

# nslookup parsing is brittle; avoid in automation
```

Notes & Pitfalls

- Output format varies; **don't parse nslookup in scripts**.
- Interactive defaults can confuse (search lists, types).
- Package: `dnsutils` / `bind-utils` / `bind9-dnsutils`.
- ANY queries may fail on public resolvers.

2026-relevant notes

Goal	Prefer
Detail	<code>dig</code>
Short human	<code>host</code> / <code>doggo</code>
Host stub	<code>resolvectl</code>
DoH/DoT	<code>doggo</code>
Legacy interactive	<code>nslookup</code>

Related Commands

- `dig` — preferred detailed client
- `host` — simple lookups
- `doggo` — modern UX
- `resolvectl` — systemd-resolved

Additional Resources

- `man nslookup`

doggo

Overview

doggo is a modern command-line **DNS client** with colorful, human-friendly output. It is an optional alternative to `dig/host/nslookup` for interactive queries. Install separately (Go binary / distro package where available). Keep `dig` for portable scripts and ubiquitous server availability.

Syntax

```
doggo [query options] NAME [TYPE]
doggo [NAME] @SERVER
```

Exact flags evolve — always check `doggo --help` for your version.

Common Options (typical)

Option	Description
NAME	Domain name to query
TYPE	Record type (A, AAAA, MX, TXT, NS, ...)
@server	Nameserver to query
-n / --nameserver	Nameserver
--strategy	e.g. random/round-robin among servers
--time	Show query time
--json	JSON output (if supported)
--short	Short answers
-4 / -6	Transport family
--tls / --https / --quic	DoT/DoH/DoQ style transports when supported

Examples with Explanations

Basic queries

```
doggo example.com
doggo example.com A
doggo example.com AAAA
doggo example.com MX
doggo example.com TXT
```

Choose resolver

```
doggo example.com @1.1.1.1
doggo example.com @8.8.8.8
doggo example.com @127.0.0.53      # systemd-resolved stub
```

Reverse / other types

```
doggo 1.1.1.1 PTR
doggo example.com NS
doggo example.com SOA
doggo _http._tcp.example.com SRV
```

Compare with dig

```
dig +short example.com A
doggo example.com A
resolvectl query example.com
```

Scripting

```
# prefer dig for stable scripting
dig +short example.com A
# doggo if JSON mode available for humans/tools
doggo example.com --json 2>/dev/null | jq .
```

Encrypted DNS (when built-in)

```
doggo example.com --tls @1.1.1.1
doggo example.com --https @https://cloudflare-dns.com/dns-query
```

Feature support depends on build.

Notes / Pitfalls

- Not preinstalled on servers — document **dig** equivalents in runbooks.
- Pretty colors can break pipes — use short/JSON/plain modes.
- System stub resolvers (127.0.0.53) may differ from public resolvers — test both when debugging.
- DNSSEC/validation visibility differs by tool; confirm with **dig +dnssec** when needed.
- DoH/DoT may be blocked on corporate networks.

2026-relevant notes

- Encrypted DNS and modern transports are the interesting differentiator vs classic **dig**.
- For fleet automation, stick to **dig/resolvectl/getent hosts**.
- Pair with **mtr/curl -v** when debugging full user-visible path issues.

Related Commands

- **dig** — standard Swiss-army DNS tool
- **host** / **nslookup** — simpler classic clients
- **resolvectl** — systemd-resolved
- **getent hosts** — NSS lookup path
- **drill** — another dig-like client (ldns)

Additional Resources

- **doggo --help**
- project docs (mr-karan/doggo)

resolvectl

Overview

resolvectl manages and queries **systemd-resolved**: DNS servers, search/route domains, LLMNR/mDNS, DNSSEC/DoT status, and the DNS cache. It replaces the older **systemd-resolve** command name on current systems. Use it to answer “what DNS is *this host* actually using?”—which can differ from public **dig @8.8.8.8** results.

Syntax

```
resolvectl [options] command [argument...]
```

Common Commands

Command	Description
status	Global + per-link resolver status
query NAME	Resolve name via resolved
query -t TYPE NAME	Specific record type
flush-caches	Drop DNS cache
reset-server-features	Reset learned server features
dns [LINK [SERVER...]]	Show/set DNS servers for a link
domain [LINK [DOMAIN...]]	Search/routing domains
default-route [LINK BOOL]	Whether link is default DNS route
statistics	Cache statistics
log-level [LEVEL]	Debug resolved
monitor	Stream resolution events (if available)

Key Use Cases

1. Debug DNS on NetworkManager / netplan / systemd-networkd hosts
2. See split-DNS (VPN vs LAN) per interface
3. Flush stale cache after record changes
4. Confirm stub listener 127.0.0.53

Examples with Explanations

Status and query

```
resolvectl status
resolvectl query example.com
resolvectl query -t MX example.com
resolvectl query -t AAAA example.com
```

Cache

```
sudo resolvectl flush-caches
resolvectl statistics
```

Per-link DNS

```
resolvectl dns
resolvectl dns eth0
resolvectl domain
ip -br link      # correlate interface names
```

One-liner recipes

```
# Compare stub vs public
resolvectl query example.com
dig +short example.com @1.1.1.1

# After VPN connect
resolvectl status | sed -n '1,80p'
resolvectl flush-caches

# Show current DNS from resolv.conf angle
cat /etc/resolv.conf
# often points to 127.0.0.53 when resolved is active
```

Notes & Pitfalls

- If **resolved** is disabled, **resolvectl** fails—use **dig** against real nameservers in `/etc/resolv.conf`.
- Setting DNS with **resolvectl dns LINK ...** may be **transient**; persistent config belongs in `netplan`, `NetworkManager`, or `.network` files.

- Ubuntu netplan frequently uses resolved’s stub; editing `/etc/resolv.conf` by hand is the wrong long-term fix when it’s a symlink.
- LLMNR/mDNS noise on home LANs can surprise; status shows whether enabled.

2026-relevant notes

- **DNS-over-TLS** may be configured via resolved (`DNSOverTLS=`)—check status for “DNS-over-TLS: yes/no”.
- Prefer `resolvectl` + `nmcli` over legacy `nm-tool`.
- Containers may not run resolved; DNS is often injected by the runtime (`/etc/resolv.conf` with cluster DNS).

Related Commands

- `dig / host / doggo` — query arbitrary servers
- `nmcli` — NetworkManager DNS
- `networkctl` — `systemd-networkd`
- `systemd-resolve` — old name (compat)

Additional Resources

- `man resolvectl`, `man systemd-resolved.service`
- freedesktop systemd resolved docs

curl

Overview

`curl` transfers data to/from URLs. It supports HTTP(S), FTP, SFTP, and many other protocols — the default tool for APIs, downloads, health checks, and quick TLS debugging. Prefer `curl` in scripts for predictable flags; use `wget` when you want recursive site mirrors.

Syntax

```
curl [options] [URL...]
```

Common Options

Option	Description
<code>-O</code>	Save using remote file name
<code>-o file</code>	Save to file
<code>-L</code>	Follow redirects
<code>-f</code>	Fail on HTTP 400 (exit non-zero)
<code>-sS</code>	Silent progress, but show errors
<code>-v / -vv</code>	Verbose (headers, TLS)
<code>-I</code>	HEAD request only
<code>-X METHOD</code>	Custom method (POST, PUT, ...)
<code>-H "K: V"</code>	Request header
<code>-d data</code>	Body (form-encoded by default)
<code>--json data</code>	JSON body + content-type (curl 7.82+)
<code>-u user:pass</code>	HTTP auth
<code>-A agent</code>	User-Agent
<code>-m secs / --max-time</code>	Overall time limit
<code>--connect-timeout secs</code>	Connect phase only
<code>-w fmt</code>	Write-out variables after transfer
<code>-k</code>	Insecure TLS (lab only)
<code>-C -</code>	Resume transfer
<code>-x proxy</code>	HTTP/SOCKS proxy
<code>-b / -c</code>	Cookie read / write jar
<code>--retry N</code>	Retry transient failures

Key Use Cases

1. Download artifacts and releases
2. Call REST APIs
3. Debug HTTP headers and TLS
4. Scripted health checks and deploys

Safety

- Prefer env vars / netrc over secrets on the command line (ps can see argv).
- -k disables certificate verification — never in production automation.
- Recursive fetchers and huge mirrors belong to wget/rcurl with bandwidth limits.

Examples with Explanations

Download

```
curl -fL -O https://example.com/file.tgz
curl -fL -o app.tgz https://example.com/file.tgz
```

-f fails on HTTP errors; -L follows redirects common on CDN links.

API GET with pretty JSON

```
curl -sS https://api.github.com/repos/jqlang/jq | jq .
```

-sS keeps scripts clean while still reporting failures.

POST JSON

```
curl -sS -X POST https://httpbin.org/post \
  -H 'Content-Type: application/json' \
  --data-binary '{"name":"ada","ok":true}' | jq .
```

newer curl:

```
curl -sS --json '{"name":"ada"}' https://httpbin.org/post | jq .
```


Headers only / method checks

```
curl -sSI https://example.com
curl -sS -o /dev/null -w '%{http_code}\n' https://example.com
```

Follow redirects, treat 404 as failure

```
curl -fsSL -o out.html https://example.com/missing || echo "failed:$?"
```

-f makes 4xx/5xx non-zero — essential in CI.

Timing metrics

```
curl -o /dev/null -sS -w \
'dns:%{time_namelookup} connect:%{time_connect} tls:%{time_appconnect} ttfb:%{time_starttransfer}'
https://example.com
```

Useful for “is it DNS, TCP, TLS, or the app?”

Auth

```
curl -sS -H "Authorization: Bearer $TOKEN" https://api.example.com/v1/me
curl -sS -u "$USER:$PASS" https://example.com/private/
curl -sS --netrc-file ~/.netrc https://example.com/private/
```

Upload file (multipart)

```
curl -fL -F "file=@./report.pdf;type=application/pdf" https://httpbin.org/post
```

Client certificate (mTLS)

```
curl -sS --cert client.pem --key client.key --cacert ca.pem https://svc.internal/health
```

Proxy and retries

```
curl -fsSL --retry 3 --retry-delay 2 -x http://proxy:8080 -O https://example.com/file
```

Show response headers + body separately

```
curl -sS -D headers.txt -o body.json https://api.example.com/v1/x
```

Understanding Output

Default is the response body on stdout and a progress meter on stderr. HTTP status does **not** affect exit code unless `-f` (or you inspect `%{http_code}`). Write-out variables expand after the transfer completes.

Notes & Pitfalls

- Quote URLs containing `&`, `?`, or shell globs.
- HTTP/2 and HTTP/3 behavior depends on how curl was built.
- Multiple URLs on one command line are sequential.
- `curl | jq` fails closed if either side fails when `set -o pipefail` is on.

Related Commands

- `wget` — download-centric alternative
- `httpie` / `xh` — friendlier human HTTP CLIs
- `jq` — JSON post-processing
- `openssl s_client` — deep TLS debug
- `ss` / `dig` — connectivity / DNS triage

Additional Resources

- `man curl`
- [Everything curl](#)

wget

Overview

wget is a non-interactive downloader for HTTP(S), FTP, and related protocols. It shines at recursive site fetches, resume, rate limiting, and simple “get this URL into a file” scripts. For complex APIs and arbitrary methods, **curl** is usually better; for bulk downloads and mirroring, **wget** is often clearer.

Syntax

```
wget [OPTIONS] [URL...]  
wget [OPTIONS] -i url-list.txt
```

Common Options

Option	Description
-O file	Write to this filename (- = stdout)
-o logfile	Write messages to log file
-c	Continue / resume partial download
-q	Quiet
-v / -nv	Verbose / non-verbose
-b	Go to background after start
-P dir	Directory prefix for saved files
-N	Timestamping (re-get only if newer)
--limit-rate=RATE	Throttle (e.g. 500k)
--timeout=SECS	Network timeout
--tries=N	Retry count (0 = infinite)
--retry-connrefused	Retry even on connection refused
-U agent	User-Agent string
--header='K: V'	Extra HTTP header
--user= / --password=	Auth (prefer netrc; see Safety)
-r	Recursive download
-l N	Recursion depth
-np	No parent (stay under URL path)
-k	Convert links for local viewing

Option	Description
<code>-p</code>	Page requisites (CSS/images for HTML)
<code>-m</code>	Mirror shorthand (<code>-r -N -l inf --no-remove-listing</code>)
<code>-A / -R</code>	Accept / reject filename patterns
<code>--spider</code>	Check existence without downloading
<code>--no-check-certificate</code>	Skip TLS verify (avoid)

Safety

- Do not put passwords on the command line (**ps** can see them). Prefer `~/.wgetrc` permissions 600 or `.netrc` with tight modes, or prompt-based tools.
- Recursive `-r` can pull far more than intended (entire sites). Use `-np`, `-l`, and `-A/-R` limits.
- `--no-check-certificate` disables TLS verification — only for lab debugging of broken certs, never production automation.
- Respect site `robots.txt` and terms; `-e robots=off` exists but is often rude or disallowed.

Examples with Explanations

Simple download (keeps remote name)

```
wget https://example.com/files/app-1.2.3.tgz
```

Save under a specific name

```
wget -O app.tgz https://example.com/files/app-latest.tgz
```

Resume interrupted download

```
wget -c https://example.com/big.iso
```

Quiet download into a directory

```
mkdir -p ~/downloads
wget -q -P ~/downloads https://example.com/tool.deb
```

Rate-limited fetch with retries

```
wget --limit-rate=1m --tries=5 --timeout=30 \  
https://example.com/dataset.tar.gz
```

Multiple URLs from a file

```
wget -i urls.txt -P ./artifacts/
```

One URL per line in `urls.txt`.

Spider: check if URL is reachable

```
wget --spider -q https://example.com/health && echo up || echo down
```

Authenticated download (prefer netrc)

```
# ~/.netrc (chmod 600):  
# machine example.com login USER password SECRET  
wget https://example.com/private/build.zip
```

Custom header (e.g. token) without logging token in shell history carefully

```
wget --header="Authorization: Bearer ${TOKEN}" \  
-O release.json https://api.example.com/v1/release
```

Still prefer env vars over pasting secrets into shell history.

One-page offline copy (HTML + assets)

```
wget -p -k -P ./mirror https://example.com/docs/intro.html
```

`-p` gets page requisites; `-k` rewrites links for local browsing.

Bounded recursive mirror of a docs tree

```
wget -r -np -nH -l 3 -P ./docs \
    https://example.com/manual/
```

- `-np` — do not ascend to parent
- `-nH` — no host-prefixed directory
- `-l 3` — depth limit

Timestamped re-sync of a single artifact

```
wget -N https://example.com/app.tgz
```

Skips download if local file is already up to date (server must send proper timestamps).

Understanding Output

Default progress shows percent, bar, speed, and ETA. `-q` silences progress (errors still appear unless fully quieted). HTTP error codes produce non-zero exit status in modern `wget` — check `$?` in scripts. Partial files remain when interrupted unless you clean them; `-c` reuses them.

Notes & Pitfalls

- `-O` with multiple URLs concatenates into one file — usually wrong; use `-P` or separate invocations.
- Recursive mode creates directory trees; know where `-P` points before filling the disk.
- Some CDNs need a browser-like User-Agent (`-U`) or extra headers; prefer fixing the URL/API over spoofing when you control the service.
- For REST APIs with POST/JSON, use `curl`; `wget` can POST (`--post-data`) but is less ergonomic.
- On Ubuntu, package is typically `wget` from `apt`; GNU `wget` 1.x is the common default (`wget --version`).

Related Commands

- `curl` — flexible transfers and APIs
- `aria2c` — multi-connection downloads (if installed)
- `apt` / `apt-get` — packages, not arbitrary URLs
- `rsync` — sync trees you already host over SSH
- `sha256sum` — verify downloads after fetch

Additional Resources

- `man wget`
- `/usr/share/doc/wget/` on Debian/Ubuntu packages

ssh

Overview

ssh (OpenSSH client) creates encrypted sessions to a remote host for shell access, command execution, port forwarding, and tunneling. Pair with **scp/sftp/rsync** for files.

Syntax

```
ssh [options] [user@]host [command]
ssh -J jump user@host
```

Common Options

Option	Description
-i key	Private key identity file
-p port	Remote port (default 22)
-L / -R / -D	Local / remote / dynamic port forwards
-N	No remote command (forward only)
-f	Background after auth
-A	Agent forwarding (use sparingly)
-J host	ProxyJump
-o Opt=Val	Config override
-v / -vv	Debug auth/connection
-t	Force TTY
-C	Compression

Key Use Cases

1. Remote administration
2. Run one-shot remote commands
3. Port forwards to private services

4. Jump-host (bastion) workflows

Safety

- Prefer keys over passwords; disable agent forwarding except when required.
- Verify host keys on first connect; investigate changed key warnings (possible MITM).
- Do not forward agent (-A) to untrusted hosts.

Examples with Explanations

Login

```
ssh alice@server.example.com
ssh -i ~/.ssh/id_ed25519 alice@server.example.com
```

Remote command

```
ssh alice@server 'hostname; uptime'
```

Custom port / options

```
ssh -p 2222 -o ConnectTimeout=5 alice@server
```

ProxyJump via bastion

```
ssh -J bastion.example.com alice@internal.example.com
# config equivalent: ProxyJump bastion.example.com
```

Local port forward (remote DB to local)

```
ssh -N -L 5432:127.0.0.1:5432 alice@server
# then connect local clients to localhost:5432
```

Dynamic SOCKS proxy

```
ssh -N -D 1080 alice@server
```

Config file (~/.ssh/config)

```
Host prod
  HostName 203.0.113.10
  User deploy
  IdentityFile ~/.ssh/prod_ed25519
  ForwardAgent no
```

```
ssh prod
```

Verbose auth debug

```
ssh -vv alice@server
```

Common Usage Patterns

Copy key (once)

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub alice@server
```

Escape sequence (client)

Enter `~ .` disconnects a wedged session; Enter `~ ?` lists escapes.

Notes & Pitfalls

- `user@host` defaults user to local username if omitted.
- Global known hosts: `~/.ssh/known_hosts`.
- Multiplexing: `ControlMaster / ControlPath` speeds repeated connections.

Related Commands

- `scp` / `sftp` — file copy
- `rsync -e ssh` — efficient sync
- `ssh-keygen` — key pairs
- `sshd` / `sshd_config` — server side

Additional Resources

- `man ssh`, `man ssh_config`

scp

Overview

`scp` (secure copy) copies files to or from a remote host over SSH. It is simple and available almost everywhere, but it is **not** a sync tool: every run re-transfers whole files. For large trees, resumes, and mirrors, prefer `rsync` over SSH. On modern OpenSSH, `scp` uses the SFTP protocol under the hood by default.

Syntax

```
scp [OPTIONS] SOURCE... DEST
# SOURCE or DEST can be: local path or [user@]host:[path]
```

Common Options

Option	Description
<code>-r</code>	Recursive (directories)
<code>-P port</code>	SSH port (capital <code>-P</code> ; not <code>-p</code>)
<code>-i identity</code>	Private key file
<code>-C</code>	Compression
<code>-p</code>	Preserve mtime and modes
<code>-q</code>	Quiet (less progress noise)
<code>-v</code>	Verbose (debug SSH/auth)
<code>-l limit</code>	Bandwidth limit in kbit/s
<code>-o Opt=Val</code>	Pass SSH option (e.g. <code>ProxyJump</code>)
<code>-3</code>	Copy between two remotes via local host

Safety

- `scp` overwrites destination files without an interactive “are you sure?” for normal paths — double-check remote paths.
- Do not embed passwords in scripts; use keys and `ssh-agent`.
- Recursively copying into the wrong remote directory can fill disks; test with a single file first.

- Prefer explicit paths (`user@host:/var/tmp/file`) over relying on remote home defaults when automating.

Examples with Explanations

Local → remote file

```
scp ./report.pdf alice@server.example.com:~/uploads/
```

Remote → local

```
scp alice@server:/var/log/nginx/access.log ./access.log
```

Directory tree

```
scp -r ./site alice@server:/var/www/
```

Copies the `site` directory as `/var/www/site` (unless you structure paths carefully).

Custom port and key

```
scp -P 2222 -i ~/.ssh/deploy_ed25519 \
  ./app.tgz deploy@server:/opt/releases/
```

Remember: `-P` for port on `scp`; `ssh/rsync` use `-p`.

Preserve times/modes

```
scp -p config.yml alice@server:/etc/myapp/config.yml
```

Via jump host (bastion)

```
scp -o ProxyJump=bastion.example.com \
  ./secret.env alice@internal:/tmp/
```

Or with OpenSSH `ProxyJump` in `~/.ssh/config` under a `Host` alias, then:

```
scp ./secret.env internal:/tmp/
```

Bandwidth-limited copy

```
scp -l 8000 large.iso alice@server:/data/
```

8000 8 Mbit/s (~1 MB/s). Units are kilobits per second.

Multiple sources to a remote directory

```
scp a.conf b.conf c.conf alice@server:/etc/myapp/
```

Remote-to-remote through your laptop

```
scp -3 alice@host-a:/data/file bob@host-b:/data/file
```

Useful when hosts cannot talk to each other directly.

Verbose debug when auth fails

```
scp -v ./file alice@server:~/
```

Understanding Output

By default `scp` shows a progress meter per file (name, percent, speed, ETA). `-q` suppresses most of that. Failures print to `stderr`; non-zero exit means the copy did not fully succeed — check partial remote files.

Notes & Pitfalls

- **Port flag case:** `scp -P 2222` vs `ssh -p 2222`. Mixing them is a common ops mistake.
- Spaces in paths need quoting on both local shell and careful remote escaping:

```
scp './My File.txt' 'alice@server:./My File.txt'
```

- `scp` is poor at resume: interrupt a multi-GB transfer and you usually start over. Use `rsync -P` instead.
- Recursive `scp` does not delete extraneous remote files (not a mirror).
- Older docs mention `scp`'s legacy RCP-based protocol; current Ubuntu OpenSSH prefers SFTP mode. If a weird old device fails, check `man scp` for `-O` (legacy mode) on your version.

- Globs expand **locally** unless quoted for the remote shell. `scp server:*.log .` may not do what you want; often better: `ssh server 'tar czf - /var/log/*.log' | tar xzf -`.

Related Commands

- `rsync -e ssh` — delta sync, dry-run, delete, resume-friendly
- `sftp` — interactive SSH file transfer
- `ssh` — remote shell / one-shot commands
- `install / install -D` — local install with mode control
- `tar + ssh` — stream many files efficiently

Additional Resources

- `man scp`
- `man ssh_config` (ProxyJump, IdentityFile, Host aliases)

rsync

Overview

rsync synchronizes files and directories locally or over a remote shell (almost always SSH). It copies only changed blocks when possible, can preserve permissions/ownership/times, and is the standard tool for backups, deploys, and large tree moves on Ubuntu servers and workstations.

Trailing slashes matter: **src/** copies the *contents* of **src** into the destination; **src** (no slash) copies the directory itself as a child of the destination.

Syntax

```
rsync [OPTIONS] SOURCE... DEST
rsync [OPTIONS] SOURCE... [USER@]HOST:DEST
rsync [OPTIONS] [USER@]HOST:SOURCE DEST
```

Common Options

Option	Description
-a	Archive mode: recursive + preserve perms, times, symlinks, devices (-rlptgoD)
-v	Verbose (list transferred files)
-z	Compress data in flight (helps over slow links; less useful on LAN/SSD)
-P	--partial --progress — keep partial files and show progress
-n / --dry-run	Show what would happen; transfer nothing
-e ssh	Use SSH as remote shell (often default nowadays)
--delete	Delete files on dest that are not on source (mirror)
--exclude=PAT	Skip matching paths
--include=PAT	Include pattern (order with exclude matters)
-x	Don't cross filesystem boundaries
-H	Preserve hard links
-A	Preserve ACLs
-X	Preserve extended attributes
--bwlimit=RATE	Throttle (e.g. 10m = 10 MB/s)

Option	Description
<code>--info=progress2</code>	Overall progress bar (GNU rsync)
<code>--checksum / -c</code>	Compare by checksum, not size+mtime
<code>-u</code>	Skip files newer on destination

Safety

- **--delete can remove large parts of the destination.** Always run with `-n` first and read the file list.
- Confirm trailing `/` intent before a mirror. `rsync -a --delete src/ dest/` is not the same as `src dest/`.
- Remote paths use `host:path` or `host:/abs/path`. A missing colon can make rsync create a local directory named like the remote host.
- Prefer SSH keys and a dedicated deploy user; avoid running rsync as root over the network unless you need ownership preservation on system trees.

Examples with Explanations

Dry-run first (always for delete or production)

```
rsync -anv --delete \
  --exclude '.git' \
  ./site/ user@server:/var/www/site/
# review the list, then drop -n:
rsync -av --delete \
  --exclude '.git' \
  ./site/ user@server:/var/www/site/
```

Local archive backup with progress

```
rsync -aHAX --info=progress2 /home/alice/ /mnt/backup/alice/
```

`-aHAX` is a solid “full metadata” local copy on Linux when ACLs/xattrs matter (e.g. home dirs, containers).

Push project over SSH

```
rsync -azP -e 'ssh -p 2222' \
  ./build/ deploy@app.example.com:/srv/app/current/
```

`-z` compresses; `-P` helps on long transfers that may interrupt.

Pull logs from a server

```
rsync -azP user@server:/var/log/nginx/ ./nginx-logs/
```

Mirror with excludes

```
rsync -a --delete \  
  --exclude 'node_modules' \  
  --exclude '.cache' \  
  --exclude '*.tmp' \  
  ./project/ /backup/project/
```

Throttled backup (shared link)

```
rsync -azP --bwlimit=5m /data/ user@offsite:/backups/data/
```

One filesystem only (skip other mounts)

```
rsync -ax / /mnt/root-backup/
```

Useful when / has bind mounts or extra disks you do not want pulled into the backup.

Checksum mode when clocks or touch times lie

```
rsync -avc ./release/ user@server:/opt/app/
```

Slower; compares content hashes instead of relying on size and mtime.

Show only what would change (itemize)

```
rsync -ani --delete ./src/ ./dest/
```

-i itemizes reasons for each transfer (permission change, size, etc.).

Understanding Output

With -v, each transferred path is printed. --info=progress2 shows aggregate bytes and rate. Dry-run (-n) still lists actions but does not write. Exit status 0 means success; non-zero usually means partial failure (permissions, vanished files, network).

Notes & Pitfalls

- **Trailing slash is the #1 footgun.** Practice with `-n` until muscle memory is solid.
- Archive mode does **not** include hard links (`-H`), ACLs (`-A`), or xattrs (`-X`) by default — add them when needed.
- `--delete` does not delete excluded files on the destination unless you also use options like `--delete-excluded` (be even more careful).
- Sparse files, open databases, and live VMs need application-consistent backups; `rsync` alone is not a snapshot tool.
- For very large trees, run from `tmux/screen` so a dropped SSH session does not kill the job; combine with `-P` to resume.

Related Commands

- `scp` — simple one-shot copy over SSH (no delta sync)
- `sftp` — interactive SSH file transfer
- `cp -a` — local recursive copy without remote/delta features
- `tar over ssh` — stream archives when `rsync` is unavailable
- `rclone` — cloud object storage sync (if installed)

Additional Resources

- `man rsync`
- `man rsyncd.conf` (daemon mode, less common for SSH workflows)

nc

Overview

nc (netcat) is a Swiss-army TCP/UDP tool: open connections, listen for inbound sockets, scan ports simply, and pipe data across the network. Ubuntu typically ships **OpenBSD netcat** as `nc` / `netcat` (`openbsd-netcat` package family; classic `netcat-traditional` may also be available). Flag details differ slightly between implementations — check `nc -h` on your host.

Syntax

```
nc [options] host port
nc -l [options] port
command | nc host port
nc host port < file
```

Common Options (OpenBSD nc on Ubuntu)

Option	Description
-l	Listen mode
-p port	Local port (with -l, often just <code>nc -l port</code>)
-u	UDP
-v	Verbose
-z	Scan mode: no data, just connect
-w secs	Connect/idle timeout
-n	Numeric only (no DNS)
-k	Keep listening after client disconnect (where supported)
-s addr	Source address
-q secs	Quit after EOF on stdin after delay
-4 / -6	IPv4 / IPv6 only

Safety

- Listening services with `nc -l` bind whatever you tell them — do not expose on public interfaces unless intended; combine with firewall rules.

- “Shell over netcat” patterns are classic attack/persistence techniques. Use only on systems you own, with authorization, for legitimate recovery labs.
- `-z` port scans against machines you do not operate may violate policy.

Examples with Explanations

Test if a TCP port is open

```
nc -vz example.com 443
nc -vz -w 3 10.0.0.10 22
```

Verbose zero-I/O connect is the operator’s quick “is something listening?” check.

HTTP GET by hand

```
printf 'GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n' \
| nc example.com 80
```

Useful when debugging virtual hosts or seeing raw headers without curl.

TLS is not built into classic nc

```
# Prefer:
openssl s_client -connect example.com:443 -servername example.com
# or curl -v https://...
```

Plain nc speaks TCP bytes; it does not negotiate TLS for you.

Simple file transfer (lab / trusted LAN)

Receiver:

```
nc -l 9000 > received.tgz
```

Sender:

```
nc -w 3 receiver.example.com 9000 < payload.tgz
```

Prefer scp/rsync for anything real (auth, encryption, integrity).

Listen and show inbound banner/data

```
nc -lvk 9000
```

Connect from another host: `nc host 9000` and type lines.

UDP probe

```
nc -u -vz -w 2 10.0.0.10 53
```

UDP “success” is weaker than TCP — lack of ICMP unreachable may still print as open/filtered depending on version.

Port range scan (light)

```
nc -vz 10.0.0.10 20-25
```

For serious inventory use `nmap`; `nc` is fine for a handful of ports.

Pipe program output to a remote listener

```
# listener:
nc -l 7000 | tar xzf -
# sender:
tar czf - ./project | nc -w 3 host 7000
```

Bind to a specific local address

```
nc -l -s 127.0.0.1 8080
```

Localhost-only listeners reduce accidental exposure.

Chatty timeout for scripts

```
if nc -z -w 2 db.internal 5432; then
    echo db-port-open
else
    echo db-port-closed
fi
```

Understanding Output

- Connect success (OpenBSD nc): often `Connection to host port [tcp/*] succeeded!` with `-v/-z`.
- Failure: connection refused (nothing listening / RST), timed out (filtered or blackholed), name resolution errors.
- In connected mode, stdin goes to the socket and socket data goes to stdout — making nc a network pipe.

Notes & Pitfalls

- **Implementation flags differ.** Scripts written for `netcat-traditional` (`-e exec`) will not match OpenBSD nc. On Ubuntu, `update-alternatives` or package choice can change behavior — verify with `nc -h` and `readlink -f $(command -v nc)`.
- Exit codes for `-z` are useful in bash if `nc -z ...; then` but confirm on your variant.
- Listening without `-k` usually exits after one client (OpenBSD). Plan for that in demos.
- IPv6 addresses need careful syntax; prefer `-6` and explicit addresses when debugging dual-stack.
- For HTTP(S) health checks, `curl -fsS -o /dev/null -w '%{http_code}\n'` is usually better than raw nc.

Related Commands

- `curl` / `wget` — HTTP(S) clients
- `openssl s_client` — TLS debugging
- `ss -tlnp` — local listeners
- `nmap` — structured port scanning
- `socat` — more powerful relay/socket toolkit (if installed)
- `telnet` — interactive TCP (legacy; nc often replaces it)

Additional Resources

- `man nc`
- `man ncat` (Nmap's netcat, if installed as separate tool)

tcpdump

Overview

tcpdump captures and displays packets on a network interface. It is the classic CLI packet sniffer for verifying traffic, debugging firewalls, and confirming “did this SYN leave the box?” On Ubuntu, install with `sudo apt install tcpdump`. Captures usually need root or `cap_net_raw` / `cap_net_admin`.

Syntax

```
sudo tcpdump [OPTIONS] [expression]
sudo tcpdump -i IFACE -nn -s0 -w file.pcap 'filter'
```

Common Options

Option	Description
-i iface	Interface (any for all, where supported)
-n	Don't resolve hostnames
-nn	Don't resolve hostnames or ports
-v / -vv	More packet detail
-c N	Exit after N packets
-s snaplen	Capture snap length (0 or 65535 = full packet)
-w file	Write raw packets to pcap file
-r file	Read packets from pcap file
-A	ASCII payload
-X / -XX	Hex + ASCII
-q	Quieter output
-e	Link-level header (MAC)
-ttt	Delta time between packets
-Z user	Drop privileges to user after opening capture
--immediate-mode	Lower latency display (when supported)

Safety

- Packet captures can include **credentials, cookies, and PII**. Restrict pcap file permissions (`chmod 600`) and delete when done.
- Capturing on production can add load; prefer tight BPF filters and `-c` limits.
- Legal/policy: only capture networks you are authorized to monitor.
- Do not publish pcaps from customer environments.

Filter expressions (BPF) essentials

Filters use libpcap syntax (same family as Wireshark capture filters):

Expression	Matches
<code>host 1.2.3.4</code>	Traffic to/from host
<code>net 10.0.0.0/8</code>	Network
<code>port 443</code>	TCP or UDP port
<code>tcp port 22</code>	SSH
<code>src / dst</code>	Direction qualifier
<code>icmp</code>	ICMP
<code>tcp[tcpflags] & tcp-syn != 0</code>	SYN packets
<code>and / or / not</code>	Boolean

Examples with Explanations

See anything on default route interface

```
ip route show default
sudo tcpdump -i eth0 -nn -c 20
```

HTTPS-related traffic for one host

```
sudo tcpdump -i any -nn 'host 203.0.113.10 and port 443'
```

DNS queries and responses

```
sudo tcpdump -i any -nn port 53
```

Only SYN packets (connection attempts)

```
sudo tcpdump -i any -nn 'tcp[tcpflags] & (tcp-syn) != 0 and tcp[tcpflags] & (tcp-ack) == 0'
```

Capture full packets to file for Wireshark

```
sudo tcpdump -i eth0 -nn -s0 -w /tmp/trace.pcap 'port 5432'  
# later, as yourself:  
sudo chown "$USER":"$USER" /tmp/trace.pcap  
chmod 600 /tmp/trace.pcap  
wireshark /tmp/trace.pcap # or tcpdump -r
```

Read a pcap back in the terminal

```
tcpdump -nn -r /tmp/trace.pcap | less  
tcpdump -nn -A -r /tmp/trace.pcap 'port 80'
```

HTTP cleartext headers (lab only)

```
sudo tcpdump -i any -nn -A 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)
```

Prefer `-w` + Wireshark for serious HTTP analysis. TLS on 443 will not show HTTP headers in cleartext.

ICMP (ping) traffic

```
sudo tcpdump -i any -nn icmp
```

Traffic involving a local port (e.g. reverse proxy)

```
sudo tcpdump -i any -nn 'port 8080'
```

Limit and timestamp deltas

```
sudo tcpdump -i eth0 -nn -c 50 -ttt host 10.0.0.5
```

Exclude SSH so your session does not flood the capture

```
sudo tcpdump -i eth0 -nn "not port 22"
```

When capturing over SSH, your interactive session traffic will otherwise dominate the output.

Understanding Output

Example TCP line:

```
12:01:02.123456 IP 10.0.0.5.54321 > 10.0.0.10.443: Flags [S], seq 123, win 64240, options [.
```

- Addresses are `ip.port` with `-nn`.
- **Flags:** S SYN, F FIN, P PSH, R RST, . ACK only.
- **length** — payload length (0 for pure handshake packets).

If you see SYNs with no SYN-ACK, the peer or a middlebox is dropping or blackholing the connection. If SYN-ACK returns but app fails, debug above TCP (TLS, HTTP, app).

Notes & Pitfalls

- Without `-s0/-s 65535`, snaplen may truncate payloads (headers usually still visible).
- Interface `any` is convenient but may not show link-layer MACs the same way as a real NIC; use a specific iface for L2 issues.
- Offloading/TSO can make on-wire sizes look odd in captures; remember NIC offload when diagnosing MSS/MTU.
- Filters are **capture** filters (BPF), not Wireshark display filters. Syntax differs from Wireshark GUI search.
- Containers: capture in the correct network namespace (`ip netns exec`, or from the host `veth`) or you will see nothing useful.
- Prefer writing to `pcap` (`-w`) for anything beyond a quick glance; `live -A` scrolls too fast and loses context.

Related Commands

- `wireshark` / `tshark` — GUI / CLI deeper analysis
- `ss -tnp` — which processes own sockets
- `iptables` / `nft` / `ufw` — firewall rules that drop traffic
- `nmap` — active port discovery (different from capture)
- `curl -v` / `openssl s_client` — application-level TLS/HTTP debug

Additional Resources

- `man tcpdump`
- `man pcap-filter` — full BPF expression reference

nmap

Command Overview

The **nmap** (Network Mapper) is a powerful open-source tool for network exploration, security scanning, and auditing. It can discover hosts and services on a network, detect operating systems, and identify potential vulnerabilities.

Syntax

```
nmap [Scan Type] [Options] {target specification}
```

Common Options

Option	Description
-sS	TCP SYN scan (default)
-sT	TCP connect scan
-sU	UDP scan
-sN	TCP NULL scan
-sF	TCP FIN scan
-sX	TCP XMAS scan
-sA	TCP ACK scan
-sW	TCP Window scan
-sM	TCP Maimon scan
-sn	Ping scan (disable port scan)
-Pn	Skip host discovery
-p	Port specification
-F	Fast scan (100 ports)
-r	Scan ports consecutively
-T<0-5>	Timing template
-sV	Version detection
-O	OS detection
-A	Enable OS detection, version detection, script scanning, and traceroute
-oN	Output normal format
-oX	Output XML format

Option	Description
-oG	Output grepable format
-v	Increase verbosity
-d	Increase debugging
--script	NSE script selection
--script-args	NSE script arguments

Key Use Cases

1. Network discovery
2. Port scanning
3. Service version detection
4. Operating system detection
5. Vulnerability assessment
6. Security auditing
7. Network inventory
8. Performance analysis

Examples with Explanations

1. Basic Scan

```
$ nmap 192.168.1.0/24
Starting Nmap 7.94 ( https://nmap.org )
Nmap scan report for 192.168.1.1
Host is up (0.0020s latency).
Not shown: 995 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
443/tcp   open  https
```

Scan entire subnet for open ports

2. Intensive Scan

```
$ nmap -A -T4 example.com
```

Aggressive scan with OS and version detection

3. Stealth Scan

```
$ sudo nmap -sS -p- example.com
```

SYN scan of all ports

4. Service Version Detection

```
$ nmap -sV -p 22,80,443 example.com
```

Detect service versions on specific ports

5. OS Detection

```
$ sudo nmap -O example.com
```

Identify operating system

Understanding Nmap

Scan Types

```
# TCP SYN Scan (Stealth)
$ sudo nmap -sS target

# TCP Connect Scan
$ nmap -sT target

# UDP Scan
$ sudo nmap -sU target

# SCTP INIT Scan
$ sudo nmap -sY target

# FIN Scan
$ sudo nmap -sF target
```

Port Selection

```
# Specific ports
$ nmap -p 80,443 target

# Port ranges
$ nmap -p 1-1000 target

# All ports
$ nmap -p- target

# Top ports
$ nmap --top-ports 100 target

# Fast scan
$ nmap -F target
```

Host Discovery

```
# Ping scan only
$ nmap -sn 192.168.1.0/24

# Skip ping
$ nmap -Pn target

# TCP SYN ping
$ nmap -PS22,80,443 target

# TCP ACK ping
$ nmap -PA22,80,443 target

# UDP ping
$ nmap -PU53 target
```

Version Detection

```
# Light version detection
$ nmap -sV --version-intensity 5 target

# Aggressive version detection
$ nmap -sV --version-all target

# With script scanning
$ nmap -sV -sC target
```


Script Scanning

```
# Default scripts
$ nmap -sC target

# Specific script
$ nmap --script=http-title target

# Script category
$ nmap --script=vuln target

# Multiple scripts
$ nmap --script=http-*,ssl-* target

# Script with arguments
$ nmap --script http-brute --script-args http-brute.path=/login target
```

Output Formats

```
# Normal output
$ nmap -oN scan.txt target

# XML output
$ nmap -oX scan.xml target

# Grepable output
$ nmap -oG scan.grep target

# All formats
$ nmap -oA scan target
```

Performance Tuning

```
# Timing templates
$ nmap -T0 target # Paranoid
$ nmap -T1 target # Sneaky
$ nmap -T2 target # Polite
$ nmap -T3 target # Normal
$ nmap -T4 target # Aggressive
$ nmap -T5 target # Insane

# Custom timing
$ nmap --min-rate 100 --max-rate 500 target
```

Advanced Techniques

```
# Fragmented packets
$ sudo nmap -f target

# Custom MTU
$ sudo nmap --mtu 24 target

# Decoy scan
$ sudo nmap -D decoy1,decoy2,ME target

# Idle scan
$ sudo nmap -sI zombie_host target

# Source port manipulation
$ sudo nmap --source-port 53 target
```

Firewall Evasion

```
# Fragment packets
$ sudo nmap -f target

# Use decoy
$ sudo nmap -D RND:10 target

# Spoof MAC
$ sudo nmap --spoof-mac Dell target

# Data length
$ sudo nmap --data-length 25 target
```

NSE Scripts Examples

```
# SSL/TLS scanning
$ nmap --script ssl-enum-ciphers -p 443 target

# Vulnerability scanning
$ nmap --script vuln target

# Brute force
$ nmap --script brute target

# Default credential check
```

```
$ nmap --script http-default-accounts target

# DNS enumeration
$ nmap --script dns-brute target
```

Best Practices

```
# Network inventory
$ nmap -sn -oX inventory.xml 192.168.1.0/24

# Security audit
$ sudo nmap -A -v -oA audit target

# Regular monitoring
$ nmap -sS -sV --open -oG monitor target

# Vulnerability assessment
$ nmap --script vuln -sV -p- target
```

Related Commands

- netcat - Network utility
- tcpdump - Packet analyzer
- wireshark - Network protocol analyzer
- hping3 - Network tool
- masscan - Mass IP port scanner

Additional Resources

- Man page: `man nmap`
- Nmap reference guide
- NSE script documentation
- Network scanning guide
- Security best practices
- Port scanning techniques

openssl

Overview

`openssl` is a Swiss-army toolkit for TLS/SSL, X.509 certificates, keys, digests, and simple encrypted files. Operators use it to inspect certificates, test TLS handshakes to a host:port, generate CSRs/self-signed certs for labs, and convert PEM/DER formats. It is not a full ACME client — prefer `certbot/acme.sh` for Let's Encrypt production issuance. Deep crypto design and application embedding belong elsewhere; this page is **operator triage**.

Syntax

```
openssl <subcommand> [options]
openssl help
openssl help s_client
```

Common subcommands: `s_client`, `x509`, `req`, `rsa`, `ec`, `pkey`, `dgst`, `rand`, `enc`, `version`.

Common Subcommands & Options

Area	Command sketch	Role
TLS probe	<code>openssl s_client -connect host:443</code>	Live handshake + cert chain
Show cert	<code>openssl x509 -in f.pem -noout -text</code>	Decode PEM certificate
CSR	<code>openssl req -new -...</code>	Certificate signing request
Digest	<code>openssl dgst -sha256 file</code>	Hash file
Random	<code>openssl rand -hex 16</code>	Random bytes
Connect SNI	<code>s_client -servername name -connect ...</code>	Virtual-host TLS

`s_client` (frequent flags)

Option	Description
<code>-connect host:port</code>	Target

Option	Description
<code>-servername name</code>	SNI (required for most vhosts)
<code>-showcerts</code>	Print full chain
<code>-starttls smtp\ imap\ ...</code>	STARTTLS protocols
<code>-alpn h2,http/1.1</code>	ALPN offer
<code>-tls1_2 / -tls1_3</code>	Pin protocol (version-dependent)
<code>-status</code>	OCSP stapling request

x509 (frequent flags)

Option	Description
<code>-in file / -out file</code>	Input/output
<code>-noout</code>	Don't emit encoded cert
<code>-text</code>	Full decode
<code>-subject -issuer -dates</code>	Short fields
<code>-fingerprint -sha256</code>	Fingerprint
<code>-inform PEM\ DER</code>	Encoding

Safety

- Never paste **private keys** into tickets/chat. Mode 600, dedicated paths.
- Self-signed lab certs are fine for experiments; browsers/clients will warn — don't disable verification in production clients casually.
- `openssl enc` password-based file encryption is easy to misuse (KDF/parameters); prefer age/GPG for human file encryption workflows unless you know the flags.
- Old TLS versions may be disabled in modern OpenSSL builds — failures can mean policy, not “host down”.

Examples with Explanations

Version and providers

```
openssl version -a
```

Confirms library version (3.x on current Ubuntu LTS).

Probe a website's certificate

```
echo | openssl s_client -servername example.com -connect example.com:443 2>/dev/null | openssl
```

echo | completes the session so `s_client` exits. Always set **SNI** (`-servername`) for shared hosting.

Full chain and leaf details

```
echo | openssl s_client -showcerts -servername example.com -connect example.com:443 2>/dev/n
echo | openssl s_client -servername example.com -connect example.com:443 2>/dev/null \
| openssl x509 -noout -text | less
```

SHA-256 fingerprint

```
echo | openssl s_client -servername example.com -connect example.com:443 2>/dev/null \
| openssl x509 -noout -fingerprint -sha256
openssl x509 -in /etc/ssl/certs/ssl-cert-snakeoil.pem -noout -fingerprint -sha256
```

Inspect a local PEM cert

```
openssl x509 -in fullchain.pem -noout -subject -issuer -dates
openssl x509 -in fullchain.pem -noout -ext subjectAltName
```

STARTTLS (mail example)

```
openssl s_client -starttls smtp -connect mail.example.com:587 -servername mail.example.com
```

Generate lab key + self-signed cert

```
openssl req -x509 -newkey rsa:2048 -keyout lab.key -out lab.crt -days 365 -nodes \
-subj '/CN=lab.local'
chmod 600 lab.key
```

For anything public-facing, use a real CA (ACME). `-nodes` leaves the key unencrypted on disk — protect the file.

CSR for an external CA

```
openssl req -new -newkey rsa:2048 -nodes -keyout app.key -out app.csr \  
-subj '/CN=app.example.com'  
openssl req -in app.csr -noout -text | head
```

PEM DER

```
openssl x509 -in cert.pem -outform DER -out cert.der  
openssl x509 -inform DER -in cert.der -out cert.pem
```

Digests (alternate to *sum tools)

```
openssl dgst -sha256 image.iso  
openssl dgst -sha512 file.bin
```

Random secrets

```
openssl rand -base64 32  
openssl rand -hex 16
```

Quick HTTPS port check (banner/handshake only)

```
echo | timeout 5 openssl s_client -connect 203.0.113.10:443 -servername app.example.com
```

Useful when curl fails and you need to see whether TLS negotiates at all.

Notes

- Ubuntu packages: `openssl` CLI; libraries via `libssl*`.
- Certificate stores: system CAs under `/etc/ssl/certs` (managed by `ca-certificates`).
- `s_client` is a debugger, not a browser — it may accept certs your app stack rejects depending on flags/defaults.
- OpenSSL 3 moved some algorithms to the “legacy” provider; rare old ciphers need extra config.
- Prefer `curl -vI --tlsv1.2` for HTTP-level checks; use `openssl s_client` for raw TLS/cert focus.

Related Commands

- `curl` — HTTPS client with cert options
- `certbot` / ACME clients — issuance/renewal
- `update-ca-certificates` — trust store
- `sha256sum` — simple file digests
- `nft` / `ufw` — not TLS; path/firewall side
- `dig` / `host` — DNS before TLS troubleshooting

Additional Resources

- `man openssl`, `man openssl-s_client`, `man openssl-x509`
- [OpenSSL documentation](#)

nmcli

Overview

`nmcli` is the CLI for **NetworkManager**. On many Ubuntu desktops/servers with NM enabled it is the supported way to manage connections, devices, and Wi-Fi—prefer connection profiles over ephemeral `ifconfig/ip addr add` for persistent config.

Syntax

```
nmcli [OPTIONS] OBJECT {COMMAND | help}
```

Objects include: `general`, `networking`, `device`, `connection`, `radio`, `monitor`, ...

Common Commands

Command	Description
<code>general status</code>	NM state
<code>device status</code>	Interfaces + state
<code>device wifi list</code>	Scan SSIDs
<code>connection show</code>	Profiles
<code>connection up/down NAME</code>	Activate/deactivate
<code>connection add ...</code>	Create profile
<code>connection modify ...</code>	Change settings
<code>connection reload</code>	Reload from disk
<code>networking off/on</code>	Global NM networking

Useful global flags: `-t` terse, `-f` fields, `-g` values only, `-p` pretty.

Key Use Cases

1. Inspect live NM state without a GUI
2. Create Ethernet/Wi-Fi profiles
3. Bring connections up/down safely
4. Script NM on headless laptops/servers

Examples with Explanations

Overview

```
nmcli general status
nmcli device status
nmcli connection show
nmcli -f NAME,UUID,TYPE,DEVICE,STATE connection show --active
```

Bounce a connection

```
nmcli connection down "Wired connection 1"
nmcli connection up "Wired connection 1"
```

Static Ethernet profile

```
nmcli connection add type ethernet ifname eth0 con-name office-static \
  ipv4.method manual \
  ipv4.addresses 192.168.1.50/24 \
  ipv4.gateway 192.168.1.1 \
  ipv4.dns "1.1.1.1,8.8.8.8"
nmcli connection up office-static
```

DHCP Ethernet

```
nmcli connection add type ethernet ifname eth0 con-name office-dhcp ipv4.method auto
```

Wi-Fi

```
nmcli device wifi list
nmcli device wifi connect "SSID" password "secret"
nmcli connection modify SSID wifi-sec.psk "new-secret"
```

Reload and DNS insight

```
nmcli connection reload
nmcli connection up <name>
nmcli dev show eth0 | rg 'IP4|IP6|DNS'
resolvectl status
```

One-liner recipes

```
# Airplane-mode-ish
nmcli radio wifi off
nmcli radio wifi on

# Script-friendly active names
nmcli -t -f NAME c show --active

# Delete a profile
nmcli connection delete office-static
```

Notes & Pitfalls

- Minimal cloud images may use **systemd-networkd** + **netplan** without NM—**nmcli** won't exist.
- Ephemeral `ip addr add` can fight NM; put durable config in profiles or netplan.
- Wi-Fi secrets end up in profile files—protect permissions.
- VPN plugins vary; check `nmcli connection show` types available.

2026-relevant notes

- Prefer **netplan** YAML on Ubuntu servers when that is the distro's source of truth—it may render NM or networkd backends.
- DoT/DNS settings increasingly appear in NM/resolved integration—verify with **resolvectl**.
- **ifconfig** is not the management API for NM systems.

Related Commands

- **ip** — kernel addresses/routes
- **resolvectl** — systemd-resolved
- **journalctl -u NetworkManager** — logs
- **nmtui** — curses UI
- **networkctl** — networkd

Additional Resources

- `man nmcli`
- NetworkManager documentation

ufw

Overview

`ufw` (Uncomplicated Firewall) is Ubuntu's friendly frontend to `netfilter`. It suits host firewalls on workstations and small servers when you want readable allow/deny rules without writing raw `iptables`/`nft` by hand. Under the hood current Ubuntu generations use `nftables` via `ufw`—you still manage policy with `ufw` commands.

Syntax

```
ufw [--dry-run] [commands]
```

Common Commands

Command	Description
<code>enable / disable</code>	Start/stop firewall
<code>status [verbose\ numbered]</code>	Show rules
<code>allow / deny / reject</code>	Add rules
<code>limit</code>	Rate-limit (anti brute-force)
<code>delete</code>	Remove by rule or number
<code>insert</code>	Insert at position
<code>reload</code>	Reload
<code>reset</code>	Wipe rules (careful)
<code>logging on\ off\ low\ medium\ high</code>	Logging
<code>app list / app info</code>	Application profiles

Key Use Cases

1. Lock down a VPS
2. Allow SSH before enabling
3. Open HTTP/HTTPS
4. Limit SSH brute-force noise

Safety

Remote sessions: always allow admin access **before** `ufw enable`. Prefer `--dry-run`. Know console/IPMI/serial recovery. Cloud security groups are a second layer outside the guest.

Examples with Explanations

Safe first enable

```
sudo ufw allow OpenSSH
# or: sudo ufw allow 22/tcp
sudo ufw --dry-run enable
sudo ufw enable
sudo ufw status verbose
```

Web and custom ports

```
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
sudo ufw allow 8080/tcp comment 'app'
```

From management network only

```
sudo ufw allow from 10.0.0.0/8 to any port 22 proto tcp
sudo ufw allow from 203.0.113.10 to any port 5432 proto tcp
```

App profiles

```
sudo ufw app list
sudo ufw allow 'Nginx Full'
sudo ufw app info 'OpenSSH'
```

Numbered delete / limit

```
sudo ufw status numbered
sudo ufw delete 3
sudo ufw limit 22/tcp
```

One-liner recipes

```
# Default policies (review before enable)
sudo ufw default deny incoming
sudo ufw default allow outgoing

# Show everything
sudo ufw status verbose

# Disable quickly
sudo ufw disable
```

Notes & Pitfalls

- Default after enable: usually **deny incoming**, allow outgoing.
- **Docker** and some CNI setups publish ports in ways that bypass or confuse UFW—verify with `ss -lntp` and cloud SG.
- IPv6 rules: ensure `IPV6=yes` in `/etc/default/ufw` when dual-stack.
- `ufw reset` clears policy—have out-of-band access.

2026-relevant notes

- Prefer **nftables**-aware mental model; `iptables-save` may show compatibility layers.
- On multi-host production, network policy often lives in cloud SG + service mesh; UFW is still fine per-VM hardening.
- RHEL-family often uses `firewalld` instead—different tool, same need to allow SSH first.

Related Commands

- `iptables` / `nft` — lower level
- `ss -lntp` — listeners before opening ports
- `journalctl` — firewall-related logs
- `fail2ban` — complementary brute-force mitigation

Additional Resources

- `man ufw`, `/usr/share/doc/ufw/`
- Ubuntu server guide firewall chapter

nft

Overview

nft is the userspace CLI for **nftables**, the modern Linux packet-filtering framework that replaces legacy **iptables** backend plumbing on current Ubuntu releases. Use it when you need structured rulesets, sets/maps, or to inspect what **ufw/iptables-nft** installed. For simple host firewalls on Ubuntu, **ufw remains the friendly default**; use **nft** for visibility, advanced policy, and routers/firewalls you own end-to-end.

Syntax

```
nft [options] commands
nft -f ruleset.nft
nft list ruleset
```

Common Commands

Command	Description
list ruleset	Dump whole configuration
list tables	Tables only
list table inet filter	One table
list chain inet filter input	One chain
add table inet filter	Create table (family ip/ip6/inet/netdev/...)
add chain ...	Create chain (with hooks for base chains)
add rule ...	Append rule
insert rule ...	Insert at position
delete rule ... handle N	Delete by handle
flush ruleset	Wipe all nft rules
flush table ... / flush chain ...	Narrow flush
-a / -n	Show handles / numeric
-j	JSON output (tooling)
-c	Check mode (with -f)
-f file	Load from file

Family **inet** covers IPv4+IPv6 in one table — preferred for new dual-stack host filters.

Safety

Remote lockout risk is identical to iptables. Always ensure SSH/admin access is allowed **before** dropping input. Prefer:

1. Console/IPMI/serial access when experimenting.
2. `nft -c -f ruleset.nft` to syntax-check.
3. Incremental changes; know how to **flush** or reboot recovery.
4. On Ubuntu desktops/servers already using **ufw**, don't fight it with ad-hoc `nft` rules unless you understand the interaction — inspect first with `nft list ruleset`.

`nft flush ruleset` clears **everything** `nft` owns — including rules installed by other tools.

Examples with Explanations

See what is active

```
sudo nft list ruleset
sudo nft list tables
sudo nft -a list chain inet filter input    # handles for deletes (names vary)
```

On UFW-enabled hosts you will typically see ufw-related tables/chains.

Minimal inet filter sketch (lab / understanding)

```
sudo nft add table inet filter
sudo nft add chain inet filter input  '{ type filter hook input priority 0; policy accept; }'
sudo nft add chain inet filter forward '{ type filter hook forward priority 0; policy drop; }'
sudo nft add chain inet filter output '{ type filter hook output priority 0; policy accept; }'

sudo nft add rule inet filter input iif lo accept
sudo nft add rule inet filter input ct state established,related accept
sudo nft add rule inet filter input tcp dport 22 accept
# only after SSH works from a second session:
# sudo nft chain inet filter input '{ policy drop; }'
```

Treat as a **teaching sketch** — production policy needs ICMP, rate limits, logging, and persistence.

Sets for many ports/IPs

```
sudo nft add set inet filter admin_v4 '{ type ipv4_addr; }'  
sudo nft add element inet filter admin_v4 '{ 203.0.113.10, 198.51.100.25 }'  
sudo nft add rule inet filter input ip saddr @admin_v4 tcp dport 22 accept
```

Sets keep rulesets readable when allow-lists grow.

Delete a rule by handle

```
sudo nft -a list chain inet filter input  
sudo nft delete rule inet filter input handle 12
```

Handles change when rules are reloaded — re-list before deleting.

Save and restore

```
sudo nft list ruleset > backup.nft  
# edit carefully...  
sudo nft -c -f backup.nft          # check  
sudo nft -f backup.nft            # apply (does not auto-flush unless file says so)
```

To replace entirely, many operators `flush ruleset` inside the file or before load — **know what you wipe**.

Persist on Ubuntu (high level)

```
# Example approach: package nftables + enable service  
sudo apt install nftables  
# put rules in /etc/nftables.conf then:  
sudo nft -c -f /etc/nftables.conf  
sudo systemctl enable --now nftables
```

If `ufw` is enabled, prefer configuring `ufw` rather than parallel conflicting `nftables.service` policies.

Coexistence checks

```
sudo ufw status verbose  
sudo nft list ruleset | head -100  
sudo iptables -L -n | head          # may be iptables-nft compat  
sudo ss -lntp
```

Confirm listeners and path: cloud security group → host nft/ufw → app.

JSON for scripts

```
sudo nft -j list ruleset | jq '.nftables | length'
```

Notes

- Prefer **one** management plane: ufw *or* raw nft *or* firewalld (RHEL) — mixing without care causes “ghost” blocks.
- Docker/Kubernetes install their own chains/tables; order and forward policy matter for published ports.
- Counter/log statements help prove hits: `counter`, `log` prefix `"nft: "`.
- IPv6: `inet` family reduces duplicated ip/ip6 tables.
- Naming: table/chain names are yours in custom rulesets; distro tools use their own.

Related Commands

- `ufw` — Ubuntu host firewall front-end
- `iptables` — legacy/compat syntax (`iptables-nft`)
- `ss -lntp` — listening sockets before opening ports
- `tcpdump` — packet proof beyond firewall counters
- `sysctl` — e.g. `net.ipv4.ip_forward` for routers
- `systemctl` — `nftables` service when used

Additional Resources

- `man nft`
- `man nftables`
- `/etc/nftables.conf` (package example)
- [nftables wiki](#)

iptables

Overview

`iptables` configures IPv4 packet filtering/NAT via **netfilter**. Many modern distros (Ubuntu 22.04+) use **nftables** underneath (`iptables-nft` compatibility). Prefer **ufw** or **nft** for new designs; still know `iptables` for legacy hosts and debugging.

Syntax

```
iptables [-t table] command chain [rule...]  
iptables-save / iptables-restore
```

Tables & common chains

Table	Chains	Role
<code>filter</code> (default)	INPUT, FORWARD, OUTPUT	Allow/deny
<code>nat</code>	PREROUTING, POSTROUTING, OUTPUT	DNAT/SNAT/MASQUERADE
<code>mangle</code>	*	TTL/mark rewrite
<code>raw</code>	PREROUTING, OUTPUT	Conntrack exceptions

Targets: ACCEPT, DROP, REJECT, LOG, RETURN, ...

Common Options

Option	Description
<code>-A/-I/-D</code>	Append / insert / delete
<code>-L -n -v</code>	List numeric verbose
<code>-F</code>	Flush chain
<code>-P chain target</code>	Default policy
<code>-s/-d</code>	Source/dest
<code>-p tcp/udp/icmp</code>	Protocol
<code>--dport/--sport</code>	Ports (with <code>-m tcp</code>)

Option	Description
<code>-i/-o</code>	In/out interface
<code>-m conntrack --ctstate</code>	State match
<code>-j</code>	Target

Safety

Remote lockout risk: open SSH before default DROP. Use console access when changing INPUT policy. Persist rules (`iptables-save`, distro-specific services) or they vanish on reboot.

Examples with Explanations

List

```
sudo iptables -L -n -v
sudo iptables -t nat -L -n -v
```

Allow SSH, HTTP; default drop (sketch)

```
sudo iptables -A INPUT -i lo -j ACCEPT
sudo iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 80 -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 443 -j ACCEPT
sudo iptables -P INPUT DROP
sudo iptables -P FORWARD DROP
# OUTPUT often left ACCEPT
```

NAT masquerade (router)

```
sudo sysctl -w net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Save (Debian/Ubuntu example)

```
sudo apt install iptables-persistent
sudo netfilter-persistent save
# or: sudo iptables-save | sudo tee /etc/iptables/rules.v4
```

Prefer UFW on Ubuntu

```
sudo ufw allow OpenSSH
sudo ufw allow 80,443/tcp
sudo ufw enable
```

Notes

- IPv6 uses `ip6tables`.
- Docker injects its own chains — order interactions carefully.
- `nft list ruleset` shows backend on nft systems.

Related Commands

- `ufw` — Ubuntu host firewall
- `nft` — nftables CLI
- `ss -lntp` — listeners
- `tcpdump` — packet proof

ifconfig

Overview

`ifconfig` configures and displays network interfaces from the legacy **net-tools** package. On modern Linux, prefer `ip` from `iproute2` (`ip addr`, `ip link`, `ip route`). This page exists for reading old docs, recovery shells that still ship `net-tools`, and muscle-memory translation to `ip`.

Many minimal images no longer install `ifconfig` by default.

Syntax

```
ifconfig [interface]
ifconfig interface [address_family] options
```

Common Options / forms

Form	Description
<code>ifconfig</code>	List up interfaces (often only “up”)
<code>ifconfig -a</code>	All interfaces including down
<code>ifconfig eth0</code>	Show one interface
<code>ifconfig eth0 up / down</code>	Bring interface up/down
<code>ifconfig eth0 192.0.2.10</code>	Set IPv4
<code>netmask 255.255.255.0</code>	
<code>ifconfig eth0 add ...</code>	Additional addresses (implementation-dependent)
<code>ifconfig eth0 hw ether</code>	Set MAC (may need down first)
<code>AA:BB:...</code>	
<code>ifconfig eth0 mtu 1400</code>	Set MTU
<code>ifconfig eth0:0 ...</code>	Alias interface style (legacy)

Examples with Explanations

Display

```
ifconfig
ifconfig -a
ifconfig eth0
```

Preferred modern equivalents

```
ip -br link
ip -br addr
ip addr show dev eth0
ip link set eth0 up
ip addr add 192.0.2.10/24 dev eth0
ip route
```

Bring up/down

```
sudo ifconfig eth0 up
sudo ifconfig eth0 down
# modern:
sudo ip link set eth0 up
```

Set address (legacy)

```
sudo ifconfig eth0 192.0.2.10 netmask 255.255.255.0
# modern CIDR:
sudo ip addr add 192.0.2.10/24 dev eth0
```

MTU

```
sudo ifconfig eth0 mtu 1400
sudo ip link set eth0 mtu 1400
```

Install on Debian/Ubuntu if missing

```
sudo apt install net-tools
```

Still prefer learning `ip`.

Notes / Pitfalls

- Output and available flags differ across Unixes; Linux net-tools is not identical to BSD.
- Changes are not persistent — use netplan/NetworkManager/systemd-networkd.
- Alias interfaces (`eth0:0`) are obsolete vs multiple addresses on one device.
- Scripts parsing `ifconfig` are fragile — use `ip -j` JSON when available.
- Wireless details: use `iw` / NetworkManager, not only `ifconfig`.

2026-relevant notes

- Teaching and certification materials may still show `ifconfig`; translate on sight to `ip`.
- Containers: `ip` is the standard; `ifconfig` often absent.
- For persistent config, never rely on either `ifconfig` or `ip` alone — use the distro network stack.

Related Commands

- `ip` — modern interface/address/route tool
- `ss` — sockets (replaces `netstat`)
- `nmcli` / `networkctl` — higher-level management
- `ethtool` — NIC hardware settings
- `ifup` / `ifdown` — distro helper scripts (legacy)

Additional Resources

- `man ifconfig` (if installed), `man ip`

netstat

Overview

`netstat` displays network connections, listening sockets, routing tables, and interface statistics from the legacy **net-tools** suite. On modern Linux, prefer **ss** (sockets) and **ip** (routes/links). `netstat` remains useful when reading older documentation or working on systems that still ship it.

Syntax

```
netstat [options]
```

Common Options

Option	Description
<code>-t / -u</code>	TCP / UDP
<code>-l</code>	Listening sockets
<code>-a</code>	All sockets
<code>-n</code>	Numeric addresses (no DNS)
<code>-p</code>	Process PID/name (may need root)
<code>-r</code>	Routing table
<code>-i</code>	Interfaces
<code>-s</code>	Statistics
<code>-c</code>	Continuous refresh
<code>-w</code>	RAW sockets
<code>-x</code>	Unix sockets
<code>-e</code>	Extended
<code>-4 / -6</code>	Address family

Classic combo: `netstat -tulpn` for listening TCP/UDP with PIDs.

Examples with Explanations

Listening services

```
netstat -tulpn
netstat -tlnp
# modern:
ss -tulpn
```

Established connections

```
netstat -tn
netstat -tp
ss -tp
```

Routing

```
netstat -rn
ip route
```

Interfaces / stats

```
netstat -i
netstat -s | less
ip -s link
```

Continuous

```
netstat -ct
watch -n1 'ss -tnp'
```

Translate old → new

Old	New
netstat -tulpn	ss -tulpn
netstat -tn	ss -tn
netstat -rn	ip route
netstat -i	ip -s link / ip -br link
netstat -s	nstat / ss -s / /proc/net

Notes / Pitfalls

- Not installed by default on many minimal distros (**net-tools** package).
- DNS reverse lookups without **-n** make output slow — prefer **-n**.
- **-p** requires privileges for others' processes.
- Output format is for humans; parse carefully or use **ss -H/ip -j**.
- Namespace-aware debugging needs **ip netns exec + ss**.

2026-relevant notes

- Muscle memory migration to **ss/ip** is complete for most ops teams; learn **netstat** only for translation.
- eBPF-based tools (**bpftool**, Pixie, etc.) go deeper than either for advanced tracing.
- Containers: check sockets **inside** the correct network namespace.

Related Commands

- **ss** — modern socket statistics
- **ip** — routes and links
- **lsof -i** — files/sockets by process
- **nstat** — network counters
- **nmap** — external port scans (authorized)

Additional Resources

- **man netstat** (if installed), **man ss**, **man ip**

ftp

Overview

`ftp` is the classic interactive client for the **File Transfer Protocol**. On modern systems it is largely superseded by **sftp/scp/rsync over SSH** and HTTPS downloads (`curl/wget`). Prefer encrypted channels whenever possible — plain FTP sends credentials and data in cleartext.

Many distros ship `ftp` as `ftp` or `tnftp` / require installing `ftp` package; some only provide `lftp` or `sftp`.

Syntax

```
ftp [options] [host]
ftp ftp://user@host/path    # URL form on some clients
```

Common Options / commands

Client option flags vary. Once connected, interactive commands include:

Command	Description
<code>open host</code>	Connect
<code>user</code>	Authenticate
<code>ls / dir</code>	List remote
<code>cd / lcd</code>	Remote / local directory
<code>get / mget</code>	Download
<code>put / mput</code>	Upload
<code>binary / ascii</code>	Transfer mode
<code>passive / passive on</code>	Passive mode (usually required today)
<code>bye / quit</code>	Exit
<code>hash</code>	Progress hashes
<code>prompt</code>	Toggle mget/mput prompts

Examples with Explanations

Interactive session

```
ftp ftp.example.com
# Name: anonymous
# Password: user@email
binary
passive
ls
get README
bye
```

Prefer SFTP instead

```
sftp user@host
scp file user@host:/path/
rsync -avP file user@host:/path/
```

lftp (better modern client)

```
lftp ftp://user@host
lftp -e 'set ftp:passive-mode true; mirror -c remote local; bye' host
```

Scripted (discouraged for secrets)

```
ftp -n host <<'EOF'
user anonymous guest@
binary
get file.bin
bye
EOF
```

Credentials in scripts leak; use SSH keys or secret stores.

Firewall note

Active FTP fails behind NATs; **passive mode** is the norm. Still painful vs SSH.

Notes / Pitfalls

- Cleartext auth and data — avoid on untrusted networks.
- FTPS (FTP over TLS) is different from SFTP (SSH); know which you need.
- Corporate scanners still find open FTP servers — disable if unused.
- ASCII mode corrupts binaries — use **binary**.
- Many public mirrors moved to HTTPS only.

2026-relevant notes

- Treat FTP as legacy interoperability only.
- Default transfer should be **sftp/rsync/curl** https.
- If you must run a server, prefer SFTP chroots or HTTPS object storage.

Related Commands

- **sftp** / **scp** / **rsync** — SSH-based transfer
- **lftp** / **ncftp** — richer FTP clients
- **curl** / **wget** — URL downloads
- **ssh** — remote shell + tunnels

Additional Resources

- `man ftp`, `man sftp`

telnet

Overview

telnet is a historic remote terminal protocol client. **Do not use telnet for remote logins** — it is unencrypted. The remaining legitimate use is as a **crude TCP debugging tool**: check whether a port accepts connections and speak simple line-based protocols by hand.

Prefer `ssh` for logins and `nc/openssl s_client/curl -v` for protocol debugging.

Syntax

```
telnet [options] host port
telnet host                # default port 23 (avoid)
```

Common Options

Option	Description
host port	Connect to TCP port
-4 / -6	Address family
-e char	Escape character
-E	Disable escape
-l user	Autologin user (legacy servers)

Interactive: escape default often `Ctrl-]`, then quit.

Examples with Explanations

Port open check

```
telnet 192.0.2.10 22
# Trying... Connected to ... → TCP open
# Connection refused → closed
# Timeout → filtered/dropped
```

Prefer:

```
nc -vz 192.0.2.10 22
# or
timeout 3 bash -c 'echo >/dev/tcp/192.0.2.10/22' && echo open
```

Speak HTTP manually

```
telnet example.com 80
# then type:
GET / HTTP/1.0
Host: example.com

# blank line ends headers
```

SMTP banner grab (authorized testing only)

```
telnet mail.example.com 25
```

Escape and quit

```
Ctrl-]
telnet> quit
```

Prefer TLS-aware tools for HTTPS/SMTPS

```
openssl s_client -connect example.com:443 -servername example.com
curl -vI https://example.com
```

Notes / Pitfalls

- Remote shell over telnet is **obsolete and unsafe**.
- “Connected” only means TCP handshake succeeded — not that the app protocol is healthy.
- Line endings and locale can confuse interactive protocol tests.
- Some minimal images omit telnet clients — install **telnet** or use **nc**.
- Never put passwords into telnet sessions on shared screen recordings.

2026-relevant notes

- Keep `nc`/`nmap`/`curl` in the toolbox; `telnet` is optional nostalgia/debug.
- Cloud security groups vs app failures: `telnet`/`nc` distinguish network path from HTTP 500s.
- IPv6: explicitly test with `-6` when dual-stack.

Related Commands

- `nc` / `ncat` — flexible network swiss army knife
- `ssh` — secure remote login
- `openssl s_client` — TLS debugging
- `curl -v` — HTTP debugging
- `nmap` — broader port scans (authorized)

Additional Resources

- `man telnet`

ssh-keygen

Overview

`ssh-keygen` creates and manages SSH key pairs, fingerprints host keys, and can update `known_hosts`. Modern default is **Ed25519**; RSA remains for legacy peers. Key management is the foundation of safe automation and admin access.

Syntax

```
ssh-keygen [options]
ssh-keygen -t ed25519 -C 'comment' -f path
ssh-keygen -lf key.pub
```

Common Options

Option	Description
<code>-t type</code>	ed25519, rsa, ecdsa, ...
<code>-b bits</code>	RSA bit length (e.g. 4096)
<code>-C comment</code>	Comment (often email)
<code>-f file</code>	Output path (private key)
<code>-N passphrase</code>	Passphrase (empty for none — automation only with care)
<code>-lf file</code>	Show fingerprint
<code>-y</code>	Derive public key from private
<code>-p</code>	Change passphrase
<code>-R host</code>	Remove host from <code>known_hosts</code>
<code>-F host</code>	Find host key in <code>known_hosts</code>
<code>-A</code>	Generate missing host keys (on servers)

Safety

- Private keys stay mode 600 and never leave your control.
- Prefer passphrases + `ssh-agent` over unprotected keys on laptops.

- Automation keys: scope with `authorized_keys` options (`from=`, `command=`, `restrict`).
- Rotate and revoke by removing the public line from `authorized_keys`.

Examples with Explanations

Create an Ed25519 key

```
ssh-keygen -t ed25519 -C 'you@example.com' -f ~/.ssh/id_ed25519
```

RSA for legacy only

```
ssh-keygen -t rsa -b 4096 -C 'legacy' -f ~/.ssh/id_rsa_legacy
```

Fingerprint

```
ssh-keygen -lf ~/.ssh/id_ed25519.pub  
ssh-keygen -E md5 -lf ~/.ssh/id_ed25519.pub # older tooling compares MD5
```

Fix / inspect `known_hosts`

```
ssh-keygen -F github.com  
ssh-keygen -R old-server.example.com
```

Public key from private

```
ssh-keygen -y -f ~/.ssh/id_ed25519 > ~/.ssh/id_ed25519.pub
```

Related Commands

- `ssh` / `ssh-copy-id` — use and install keys
- `ssh-agent` / `ssh-add` — unlock keys in memory
- `chmod` — enforce 700 on `~/.ssh`, 600 on private keys

Additional Resources

- `man ssh-keygen`
- OpenSSH release notes on default algorithms

ssh-copy-id

Overview

ssh-copy-id installs your public key on a remote account's `~/.ssh/authorized_keys` with safer permissions than hand-copying. Prefer it over pasting keys via ad-hoc `echo >>` when the remote still allows password auth for bootstrap.

Syntax

```
ssh-copy-id [options] [user@]host
ssh-copy-id -i key.pub user@host
```

Common Options

Option	Description
<code>-i key</code>	Public key file (default: first available <code>id_*.pub</code>)
<code>-p port</code>	SSH port
<code>-o Opt=Val</code>	Pass through to ssh
<code>-f</code>	Force add without duplicate check (implementation-dependent)
<code>-n</code>	Dry-run (where supported)

Safety

- Still authenticates once with the remote password (or existing key).
- Review remote `authorized_keys` after install.
- Do not use world-writable home or `.ssh` directories — sshd will ignore keys.

Examples with Explanations

Default identity

```
ssh-copy-id alice@server.example.com  
ssh alice@server.example.com
```

Explicit key and port

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub -p 2222 alice@server.example.com
```

Via ProxyJump

```
ssh-copy-id -o ProxyJump=bastion.example.com alice@internal
```

Manual equivalent (when tool missing)

```
cat ~/.ssh/id_ed25519.pub | ssh alice@server \  
'umask 077; mkdir -p ~/.ssh; cat >> ~/.ssh/authorized_keys'
```

Notes & Pitfalls

- Multiple keys: specify `-i` so the wrong identity is not installed.
- Some hardened images disable passwords — need `console/cloud-init` to place the first key.
- SELinux contexts on `~/.ssh` can block access until restored.

Related Commands

- `ssh-keygen` — create keys
- `ssh` — connect
- `getent passwd` — confirm remote home path

Additional Resources

- `man ssh-copy-id`

sftp

Overview

sftp is the interactive SSH file-transfer client (SFTP subsystem). Prefer it over legacy **ftp**/**telnet** for remote file work; for scripted bulk sync, prefer **rsync** or **scp**/**sftp** batch modes. Uses the same auth and config as **ssh**.

Syntax

```
sftp [options] [user@]host
sftp [options] -b batchfile [user@]host
```

Common Options

Option	Description
-i key	Private key
-P port	Port (note capital P , unlike ssh 's -p)
-o Opt=Val	ssh options (ProxyJump , ...)
-b file	Batch commands from file
-a	Attempt to continue interrupted downloads (where supported)
-r	Recursive for get / put in modern OpenSSH

Examples with Explanations

Interactive session

```
sftp alice@server.example.com
# sftp> ls
# sftp> get remote.txt
# sftp> put local.txt /var/tmp/
# sftp> bye
```


Batch download

```
cat > /tmp/sftp.batch <<'EOF'
cd /var/backups
lcd /tmp
get -r weekly/
bye
EOF
sftp -b /tmp/sftp.batch alice@server
```

Jump host

```
sftp -o ProxyJump=bastion.example.com alice@internal
```

Non-interactive one-shot via ssh

```
ssh alice@server 'cat /etc/os-release' > os-release.txt
rsync -av alice@server:/var/www/ ./www-mirror/
```

Often clearer than SFTP batch for automation.

Notes & Pitfalls

- Port flag is `-P` for sftp/scp historical reasons.
- Recursive behavior and resume support vary by OpenSSH version.
- File modes/ownership on put depend on remote umask and SSH config.

Related Commands

- `scp` — simple remote copy
- `rsync` — efficient sync
- `ssh` — remote shell
- `rclone` — multi-cloud sync if installed

Additional Resources

- `man sftp`
- `man sftp-server`

ethtool

Overview

`ethtool` queries and controls **E**thernet **N**IC settings: link speed/duplex, offloads, ring parameters, and driver info. Use it when `ip link` shows a device but performance or link negotiation looks wrong. Requires root for most changes.

```
sudo apt install ethtool
```

Syntax

```
ethtool [options] devname
ethtool -s devname ...      # change settings
```

Common Options

Option	Description
<i>(dev)</i>	Show basic link settings
<code>-i</code>	Driver and firmware info
<code>-k</code>	Offload features
<code>-K</code>	Change offloads
<code>-S</code>	NIC/driver statistics
<code>-g / -G</code>	Ring buffer get/set
<code>-s</code>	Change speed/duplex/autoneg/wol
<code>-p</code>	Identify port (blink LED)

Safety

- Forcing speed/duplex incorrectly causes silent poor performance or link down. Prefer autoneg when the peer supports it.
- Offload changes can fix or break checksum issues — test with care on production.

Examples with Explanations

Link status

```
ethtool eth0
# Speed, Duplex, Auto-negotiation, Link detected
```

Driver info

```
ethtool -i eth0
```

Statistics

```
ethtool -S eth0 | egrep -i 'err|drop|crc|miss'
```

Identify physical port

```
sudo ethtool -p eth0 5          # blink ~5 seconds
```

Offloads

```
ethtool -k eth0 | head
sudo ethtool -K eth0 gro off    # example; know why you toggle
```

Persist settings

Use distro network config (netplan, NetworkManager, systemd-networkd) or `ethtool` udev/systemd oneshot units — bare `ethtool -s` does not survive reboot alone.

Related Commands

- `ip link / ip -s link` — state and counters
- `nmcli` — NetworkManager
- `lspci` — hardware identity
- `dmesg` — driver attach messages

Additional Resources

- `man ethtool`

firewall-cmd

Overview

`firewall-cmd` is the client for **firewalld**, the dynamic firewall manager on Fedora, RHEL, CentOS Stream, and many related systems. Zones and services abstract raw `nftables`/`iptables` rules. On Ubuntu, prefer `ufw` or direct `nft` unless you installed `firewalld` intentionally.

```
sudo dnf install firewalld
sudo systemctl enable --now firewalld
```

Syntax

```
firewall-cmd [options]
```

Common Options / Patterns

Action	Example
State	<code>firewall-cmd --state</code>
Active zones	<code>--get-active-zones</code>
List zone	<code>--list-all / --zone=public --list-all</code>
Allow service	<code>--add-service=http --permanent</code>
Allow port	<code>--add-port=8080/tcp --permanent</code>
Reload	<code>--reload</code>
Panic	<code>--panic-on</code> (blocks all — careful)

Safety

- Always pair `--permanent` with `--reload` (or use runtime-only for experiments).
- Locking yourself out over SSH: ensure `ssh` service is allowed in the active zone before disconnecting.
- Panic mode is a last resort.

Examples with Explanations

Inventory

```
sudo firewall-cmd --state
sudo firewall-cmd --get-active-zones
sudo firewall-cmd --list-all
```

Allow HTTP permanently

```
sudo firewall-cmd --permanent --add-service=http
sudo firewall-cmd --reload
sudo firewall-cmd --list-services
```

Open a custom port

```
sudo firewall-cmd --permanent --add-port=9090/tcp
sudo firewall-cmd --reload
```

Rich rule example

```
sudo firewall-cmd --permanent --add-rich-rule='rule family="ipv4" source address="10.0.0.0/8'
sudo firewall-cmd --reload
```

Runtime vs permanent

```
sudo firewall-cmd --add-service=http           # runtime only
sudo firewall-cmd --runtime-to-permanent       # promote
```

Notes & Pitfalls

- Interfaces bind to zones (`--zone=public --change-interface=eth0`).
- Conflicts with manual `nft`/`iptables` rules are common — pick one manager.
- Container ports still need host firewall + publish rules.

Related Commands

- `ufw` — Ubuntu’s simple frontend
- `nft` / `iptables` — low-level
- `ss` — verify listeners
- `nmcli` — interface/zone integration on NM systems

Additional Resources

- `man firewall-cmd`
- `firewalld` documentation (zones and services)

Scheduling

Intro

Intro

Run work later or periodically. Classic `cron/crontab/anacron/at` remain everywhere; on `systemd` hosts also consider timers (`systemctl list-timers`, `systemd-run`).

Commands in this part

Command	Role
<code>cron</code>	<code>cron</code> is the classic daemon that runs scheduled jobs from per-user and system crontab files.
<code>crontab</code>	<code>crontab</code> installs, lists, and removes per-user crontab files that the <code>cron</code> daemon executes.
<code>anacron</code>	<code>anacron</code> runs periodic commands on machines that are not on 24/7 — laptops, desktops, and intermittent VMs.
<code>at</code>	<code>at</code> schedules a one-shot command to run once at a future time.

Suggested starting points

1. User jobs: `crontab -e` / `crontab -l`.
2. Daemon and system tables: `cron`, `/etc/cron.d`.
3. Machines that sleep: `anacron`.
4. One-shot future jobs: `at`.
5. Modern alternative: `systemd` timers (Services and runtime).

Related parts

- Services and runtime — `systemd-run`, timers
- Logging — capture job output with `logger/journalctl`
- Shell commands — reliable one-liners for jobs

Continue with the individual command pages in this part.

cron

Overview

cron is the classic daemon that runs scheduled jobs from per-user and system crontab files. On modern Ubuntu, **systemd timers** are often a better fit for service-integrated jobs, but cron remains everywhere: shared hosting, legacy scripts, and simple user schedules. This page covers how the daemon interprets schedules and where jobs live; day-to-day editing is usually via **crontab**.

Syntax

Jobs are **not** started as `cron ...`. You install lines into a crontab; **cron** reads them.

Crontab schedule fields:

```
* * * * * command

    day of week (0-7, Sun=0 or 7; some versions allow names)
    month (1-12)
    day of month (1-31)
    hour (0-23)
    minute (0-59)
```

System drop-in files under `/etc/cron.d/` use an **extra username field** after the schedule:

```
* * * * * root /usr/local/bin/job.sh
```

Common schedule patterns

Expression	Meaning
<code>*/15 * * * *</code>	Every 15 minutes
<code>0 * * * *</code>	Top of every hour
<code>0 3 * * *</code>	Daily at 03:00
<code>0 9 * * 1-5</code>	Weekdays 09:00
<code>0 0 1 * *</code>	Monthly, 1st at midnight
<code>@reboot</code>	Once at cron startup (after boot)
<code>@daily / @weekly / @hourly</code>	Nicknames (implementation-specific paths)

Special characters: `*` any, `,` list, `-` range, `/` step. Extensions like `L`, `W`, `#` are **not** portable — stick to the five standard fields unless you know your cron flavor.

Key Use Cases

1. Periodic backups and reports
2. Log or temp cleanup
3. Polling scripts that are not worth a long-running daemon
4. User-level schedules without writing systemd units

Safety

- Jobs run with a **minimal environment** — always use absolute paths and set `PATH`/`SHELL` in the crontab when needed.
- Mail/`MAILTO` can flood or leak output; redirect `stdout`/`stderr` intentionally.
- Never put secrets on the command line in a world-readable system crontab; use root-owned scripts with `0600` or secret managers.
- Overlapping long jobs: guard with `flock` so two copies do not run at once.

Examples with Explanations

User crontab (via crontab tool)

```
crontab -l
crontab -e
```

`cron` executes the commands as that user. See the `crontab` chapter for flags.

Example job lines

```
# m h dom mon dow command
0 3 * * * /usr/local/bin/backup.sh >>/var/log/backup.log 2>&1
*/15 * * * * /usr/bin/flock -n /tmp/poll.lock /usr/local/bin/poll.sh
@reboot /usr/local/bin/start-tunnel.sh
```

Redirects capture output; `flock -n` skips a run if the previous still holds the lock.

System-wide locations (Ubuntu)

Path	Role
/etc/crontab	System crontab (has user field)
/etc/cron.d/*	Drop-in packages/admin jobs (user field)
/etc/cron.daily/ etc.	Scripts run by run-parts on a schedule
/var/spool/cron/crontabs/	Per-user crontabs (do not edit by hand)

```
ls /etc/cron.d
ls /etc/cron.daily
```

Environment inside cron

```
SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MAILTO=you@example.com
0 4 * * * /usr/local/bin/nightly.sh
```

Variables at the top of a crontab apply to later lines. `HOME` is usually the user's home; do not assume interactive aliases.

Debugging “it didn't run”

```
# Is cron running?
systemctl status cron          # Debian/Ubuntu package name
systemctl status crond        # some RHEL-like systems

# Logs (paths vary)
journalctl -u cron -n 50 --no-pager
grep CRON /var/log/syslog | tail

# Syntax / permission issues
crontab -l
ls -l /usr/local/bin/nightly.sh # must be executable; readable by the job user
```

Common failures: relative paths, missing `+x`, script expects a TTY, job runs as wrong user, anacron vs laptop sleep (see **anacron**).

Prefer systemd timers when...

```
systemctl list-timers --all
# unit pair: foo.service + foo.timer
```

Timers give dependency ordering, better logging via the journal, and clearer failure states. Use cron when you need a portable one-liner or cannot install units.

Understanding Output

Successful jobs are silent unless they print. By default many crons mail any stdout/stderr to the user (MAILTO). If mail is unconfigured, output may vanish into a local mailbox or be discarded — **always log explicitly** for important jobs.

Notes & Pitfalls

- Day-of-month **and** day-of-week both set: meaning is “either matches” on Vixie/cronie (easy to schedule more often than intended).
- Timezone is the system timezone (`timedatectl`); DST transitions can skip or double an hour.
- Laptops/desktops asleep at the scheduled minute **miss** the job (use `anacron` or timers with `Persistent=true`).
- Percent signs `%` in commands are special in some crons (newline to stdin) — escape as `\%`.
- SELinux/AppArmor may deny scripts that work interactively.

Related Commands

- `crontab` — install and list user crontabs
- `anacron` — catch up periodic jobs on machines that sleep
- `at` — one-shot future jobs
- `systemctl` / `timers` — modern alternative
- `flock` — mutual exclusion for long jobs
- `logger` — send job messages to syslog/journal

Additional Resources

- `man cron` / `man 5 crontab`
- crontab.guru — schedule explainer

crontab

Overview

`crontab` installs, lists, and removes **per-user** crontab files that the `cron` daemon executes. Prefer `crontab -e` over editing spool files by hand. System-wide jobs live under `/etc/cron.d` and friends — see the `cron` chapter.

Syntax

```
crontab [-u user] file
crontab [-u user] [-l | -r | -e] [-i]
```

Common Options

Option	Description
<code>-l</code>	List current crontab
<code>-e</code>	Edit current crontab in <code>\$EDITOR</code> / <code>\$VISUAL</code>
<code>-r</code>	Remove entire crontab
<code>-i</code>	Prompt before <code>-r</code>
<code>-u user</code>	Operate on another user's crontab (root)

Key Use Cases

1. Schedule periodic tasks as your own user
2. Install a crontab from a file in config management
3. Audit what a user has scheduled
4. Clean up obsolete jobs safely

Safety

- `crontab -r` deletes **all** of that user's jobs without a backup — use `-i` or list first.
- Installing from a file replaces the whole crontab; keep the source in git.
- Root using `-u` affects another account's automation — verify the user name.

Examples with Explanations

List and edit

```
crontab -l
EDITOR=vim crontab -e
```

Empty crontab may print no crontab for user (exit status can be non-zero depending on version).

Install from a file (idempotent in config mgmt)

```
cat > /tmp/my.cron <<'EOF'
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin
MAILTO=""
*/15 * * * * /usr/local/bin/poll.sh >>$HOME/logs/poll.log 2>&1
0 3 * * * /usr/local/bin/backup.sh >>$HOME/logs/backup.log 2>&1
EOF
crontab /tmp/my.cron
crontab -l
```

Redirects and absolute paths avoid “worked in SSH, failed in cron”.

Append a line carefully

```
(crontab -l 2>/dev/null; echo '0 4 * * 0 /usr/local/bin/weekly.sh') | crontab -
```

Race-prone if two installers run at once — prefer full-file installs from CM.

Remove with confirmation

```
crontab -l > ~/crontab.backup.$(date +%F)
crontab -i -r
```

Always backup before delete.

Root manages another user

```
sudo crontab -u deploy -l
sudo crontab -u deploy -e
```

Validate schedule fragments

```
# After editing, re-list and reason about times:
crontab -l
# Cross-check system timers too:
systemctl list-timers --all
```

Common job examples

```
# every 15 minutes
*/15 * * * * /usr/bin/flock -n /tmp/poll.lock /usr/local/bin/poll.sh

# weekdays 09:00
0 9 * * 1-5 /usr/local/bin/report.sh

# after reboot
@reboot /usr/local/bin/start-tunnel.sh
```

See `cron` for field semantics, environment, and debugging.

Understanding Output

`crontab -l` prints the file body (comments and environment assignments included). Editors invoked by `-e` must exit zero for the new file to be installed; syntax is checked lightly — a bad command still “installs”.

Notes & Pitfalls

- Per-user crontabs do **not** include a username field (unlike `/etc/crontab`).
- `%` is special in some cron implementations — escape as `\%` in commands.
- SELinux may block scripts that work in an interactive shell.
- Desktop users: session-dependent jobs often need systemd user timers + lingering, not cron.
- `crontab -e` uses `select-editor` on first run (Debian/Ubuntu).

Related Commands

- `cron` — daemon and system locations
- `anacron` — catch-up on machines that sleep
- `at` — one-shot schedules
- `systemctl list-timers` — systemd alternative
- `flock` — prevent overlapping runs

Additional Resources

- `man 1 crontab`
- `man 5 crontab`
- crontab.guru

anacron

Overview

anacron runs periodic commands on machines that are **not on 24/7** — laptops, desktops, and intermittent VMs. Unlike **cron**, which expects the system to be up at a specific minute, **anacron** ensures daily/weekly/monthly jobs run when the machine is powered on after a missed window.

On many modern distros, **anacron** is integrated with **systemd timers** and `/etc/cron.{daily,weekly,monthly}` via **cron/anacron** packages.

Syntax

```
anacron [options] [job]
anacron -T # test tab syntax (when supported)
```

Common Options

Option	Description
-s	Serialize jobs; run sequentially
-f	Force run ignoring timestamps
-n	Run jobs now, ignore delays
-d	Debug to stderr; don't background
-q	Quiet with -d
-t file	Use alternate anacrontab
-S dir	Spool directory for timestamps
-u	Update timestamps without running

Configuration

Primary table: `/etc/anacrontab`

Typical fields:

period	delay	job-id	command
1	5	cron.daily	run-parts --report /etc/cron.daily
7	10	cron.weekly	run-parts --report /etc/cron.weekly
@monthly	15	cron.monthly	run-parts --report /etc/cron.monthly

Timestamps live under `/var/spool/anacron/`.

Examples with Explanations

Status / integration

```
cat /etc/anacrontab
ls /var/spool/anacron/
systemctl status anacron.service anacron.timer 2>/dev/null
systemctl list-timers | grep -i cron
```

Force a run (careful)

```
sudo anacron -f -d
sudo anacron -n -d cron.daily
```

`-d` keeps output in the foreground for debugging.

Update timestamps only

```
sudo anacron -u
```

Marks jobs as done without executing — use sparingly.

Put a job in daily.d style

```
sudo install -m 755 /usr/local/sbin/backup-homes /etc/cron.daily/backup-homes
# anacron/cron.daily will pick it up on next daily pass
```

Prefer systemd timer for new work

```
# example unit pair
# /etc/systemd/system/mybackup.service
# /etc/systemd/system/mybackup.timer
sudo systemctl enable --now mybackup.timer
```

```
systemctl list-timers
```

Notes / Pitfalls

- Random **delay** field prevents thundering herds when many laptops resume together.
- Jobs must be safe to run after multi-day gaps (catch-up logic).
- Overlap: don't also schedule the same heavy job at a fixed cron minute without coordination.
- Spool timestamps require writable `/var`; read-only root systems need alternate design.
- Logging often via syslog/journal — check `journalctl -u anacron`.

2026-relevant notes

- systemd timers with `Persistent=true` cover many anacron use cases on modern hosts.
- Still relevant for classic `/etc/cron.daily` ecosystems on Debian/Ubuntu desktops.
- Cloud fleets that auto-scale should use durable queues/schedulers, not host-local anacron alone.

Related Commands

- `cron` / `crontab` — time-based schedules
- `systemd-run` / `.timer` units — modern scheduling
- `run-parts` — run scripts in a directory
- `journalctl` — logs
- `at` — one-shot future commands

Additional Resources

- `man anacron`, `man anacrontab`

at

Overview

at schedules a **one-shot** command to run once at a future time. Unlike **cron** (recurring) or **systemd** timers (rich units), **at** is ideal for “run this once tonight” admin tasks. Companion tools: **atq** (queue), **atrm** (remove), **batch** (run when load is low).

Requires **atd** service running and permission via `/etc/at.allow` / `/etc/at.deny`.

Syntax

```
at [-q queue] [-f file] [-m] TIME
atq
atrm JOB...
batch
```

Common Options

Option	Description
<code>-f file</code>	Read job from file
<code>-m</code>	Mail when done (even if no output)
<code>-M</code>	Never mail
<code>-q queue</code>	Queue letter (a default; higher letters later)
<code>-t time</code>	Time as <code>[[CC]YY]MMDDhhmm[.ss]</code>
<code>-v</code>	Show time job will run
<code>-c job</code>	Cat job contents
<code>-l</code>	Alias for atq on some systems
<code>-d</code>	Alias for atrm on some systems

Time expressions

```
now + 30 minutes
now + 2 hours
teatime          # 16:00
noon
midnight
23:30
10am tomorrow
4pm + 3 days
```

Examples with Explanations

Interactive job

```
at now + 1 hour
# then type commands, finish with Ctrl-D
echo "hello from at" >> /tmp/at.out
systemctl restart demo
^D
```

One-liner from pipe

```
echo 'systemctl restart nginx' | sudo at now + 5 minutes
echo 'wall "maintenance over"' | at 22:00
```

From file

```
cat > /tmp/job.sh <<'EOF'
#!/bin/bash
/usr/local/bin/backup.sh
EOF
at -f /tmp/job.sh 2am tomorrow
```

Queue management

```
atq
at -c 7          # show job 7 script
atrm 7
```

batch

```
echo '/usr/local/bin/heavy-index' | batch
```

Runs when system load allows (implementation-defined).

Ensure service

```
systemctl status atd  
sudo systemctl enable --now atd
```

Notes / Pitfalls

- Environment is a snapshot — don't assume your interactive aliases; use absolute paths.
- Output mailed to user if local mail works; on servers mail may be undelivered — redirect to files.
- Permissions: not all users may use `at` (allow/deny files).
- Timezone is system timezone (`timedatectl`).
- Not a substitute for durable job queues in multi-host apps.

2026-relevant notes

- Prefer `systemd-run --on-calendar= / --on-active=` for modern one-shots with cgroups/logging.
- `at` remains handy on classic servers for quick deferred restarts.
- Containers often lack `atd` — use external schedulers.

Related Commands

- `atq` / `atrm` / `batch`
- `cron` / `crontab`
- `systemd-run` / `timers`
- `sleep` — simple delay in a live shell
- `wall` — broadcast messages

Additional Resources

- `man at`, `man atd`

Logging

Intro

Intro

Write and maintain traditional logs: kernel ring buffer, syslog-style messages, and rotation. On systemd systems, pair these with `journalctl` from Services and runtime.

Commands in this part

Command	Role
<code>dmesg</code>	<code>dmesg</code> prints the kernel ring buffer: hardware detection, driver messages, OOM kills, I/O errors, and other...
<code>logger</code>	<code>logger</code> writes messages to the system log from the shell or scripts.
<code>logrotate</code>	<code>logrotate</code> rotates, compresses, and retires log files so they do not fill disks.

Suggested starting points

1. Kernel messages: `dmesg` (and `journalctl -k`).
2. Emit a message: `logger`.
3. Manage growth: `logrotate`.

Related parts

- Services and runtime — `journalctl`
- Text and pipes — mine log files with `grep/jq`
- Archives and compression — `zgrep` on rotated `.gz` logs

Continue with the individual command pages in this part.

dmesg

Overview

`dmesg` prints the kernel ring buffer: hardware detection, driver messages, OOM kills, I/O errors, and other kernel-level events. First stop when a device doesn't appear, a disk fails, or the system OOMs. On systemd systems, the same stream is also available via `journalctl -k`.

Syntax

```
dmesg [options]
```

Common Options

Option	Description
<code>-H</code>	Human-readable timestamps
<code>-T</code>	Absolute human timestamps
<code>-k</code>	Kernel messages only
<code>-u</code>	Userspace messages (facility filter)
<code>-w / --follow</code>	Wait for new messages
<code>-W</code>	Follow after clearing wait logic variants
<code>-l level</code>	Filter by level
<code>-f facility</code>	Filter by facility
<code>-n level</code>	Set console log level
<code>-C / -c</code>	Clear / read-and-clear
<code>-x</code>	Decode facility/level prefixes
<code>-e</code>	Local time / reltime variants (see man)
<code>--color</code>	Colorize

Examples with Explanations

Recent kernel messages

```
dmesg | less
dmesg -T | less
dmesg -H
```

Follow live (plug hardware)

```
dmesg -w
# or
journalctl -kf
```

Errors and warnings

```
dmesg -T -l err,warn
dmesg -T | grep -iE 'error|fail|oom|i/o'
```

OOM investigation

```
dmesg -T | grep -i 'killed process'
journalctl -k | grep -i oom
```

USB / disk events

```
dmesg -T | tail -50
dmesg -T | grep -i usb
dmesg -T | grep -i sd
```

Clear buffer (careful)

```
sudo dmesg -C
```

Usually unnecessary; prefer filtering by time.

Permissions

```
# unprivileged may see restricted buffer on modern kernels
sudo dmesg
# sysctl kernel.dmesg_restrict
```

Notes / Pitfalls

- Ring buffer is finite — old messages drop; use journal persistence for history.
- Timestamps with `-T` depend on timekeeping across suspend/boot.
- Don't spam `dmesg -c` on production; you lose context for others.
- Userspace app logs are **not** here — use `journalctl` / `syslog`.
- Containers may not expose full host kernel messages.

2026-relevant notes

- `journalctl -k -b` for current boot kernel logs is often more convenient with persistent journal.
- Hardware still fails the same way — `dmesg` remains essential on bare metal.
- PSI and detailed MM logs complement OOM `dmesg` lines on modern kernels.

Related Commands

- `journalctl -k` — kernel via journal
- `udevadm` — device events
- `lspci` / `lsusb` — hardware lists
- `smartctl` — disk health
- `sysctl` — kernel parameters including log levels

Additional Resources

- `man dmesg`

logger

Overview

`logger` writes messages to the system log from the shell or scripts. It is the command-line interface to the syslog/journal stack — useful for marking deploy steps, cron output, and ad-hoc audit notes that should land next to service logs.

On systemd systems, messages typically appear in the **journal** (`journalctl`) and may also be forwarded to classic syslog daemons.

Syntax

```
logger [options] [message]
echo message | logger [options]
```

Common Options

Option	Description
<code>-p priority</code>	Facility.level e.g. <code>user.notice</code> , <code>local0.warning</code>
<code>-t tag</code>	Tag (defaults often to user name)
<code>-i</code>	Include PID in the tag
<code>-s</code>	Also write to stderr
<code>-f file</code>	Log each line of file
<code>-n host</code>	Remote syslog host (when supported)
<code>-P port</code>	Remote port
<code>-u socket</code>	Log socket path
<code>--journald</code>	Send via journal fields (newer util-linux)
<code>-e</code>	Skip empty lines

Examples with Explanations

Simple messages

```
logger "deploy started"
logger -t deploy -i "step 1 complete"
logger -p local0.info "backup ok"
```

From scripts

```
#!/bin/bash
tag=mybackup
log() { logger -t "$tag" -p user.notice "$*"; }
log "starting"
# ...
log "finished"
```

Pipe multi-line output

```
df -h | logger -t diskcheck
mycommand 2>&1 | logger -t mycommand -i
```

Priority levels

```
logger -p user.debug "debug detail"
logger -p user.warning "watch this"
logger -p user.err "failed to mount backup"
```

Facility.level pairs control routing in rsyslog/syslog-ng; journal stores priority as well.

View on systemd

```
logger -t demo "hello journal"
journalctl -t demo -n 5
journalctl -t demo -f
```

stderr mirror

```
logger -s -t demo "visible in console and log"
```

Structured-ish with journald (when supported)

```
logger --journald <<EOF
MESSAGE=deployment finished
PRIORITY=5
DEPLOY_ID=42
EOF
```

Notes / Pitfalls

- Rate limiting may drop floods of messages (rsyslog/journald).
- Remote logging needs network syslog config; `-n` alone may be insufficient on hardened hosts.
- Don't log secrets (tokens, passwords).
- Cron: prefer logging explicitly; cron mails can be unreliable.
- Priorities only help if collectors honor them.

2026-relevant notes

- Prefer `journalctl` for local investigation; ship logs with vector/fluent-bit/alloy for fleets.
- `systemd-cat` is an alternative that attaches stdout to the journal with unit-like metadata.
- For apps, native journal/syslog libraries beat wrapping everything in `logger`.

Related Commands

- `journalctl` — query journal
- `systemd-cat` — pipe to journal
- `rsyslogd` / `syslog-ng` — classic daemons
- `dmesg` — kernel ring buffer
- `wall` — broadcast to ttys (not syslog)

Additional Resources

- `man logger`

logrotate

Overview

logrotate rotates, compresses, and retires log files so they do not fill disks. On Ubuntu it is driven by a **systemd timer** or cron, reading `/etc/logrotate.conf` plus drop-ins in `/etc/logrotate.d/`. Application logs you add should get a file under `/etc/logrotate.d/`. Journald logs are **not** managed here — use `journalctl --vacuum=*` for the journal.

Syntax

```
logrotate [options] configfile
sudo logrotate -d /etc/logrotate.conf
sudo logrotate -f /etc/logrotate.d/rsyslog
```

Common Options

Option	Description
-d	Debug/dry-run — no changes
-f	Force rotation even if not due
-v	Verbose
-s statefile	Alternate state file (default <code>/var/lib/logrotate/status</code>)
--skip-state-cleanup	Keep status entries for vanished logs (version-dependent)

Common config directives

Directive	Meaning
daily / weekly / monthly / hourly	Time-based rotation
size 100M / maxsize 100M	Size-based (see man for size vs maxsize)
rotate N	Keep N old logs
compress / delaycompress	gzip (delay: skip most recent rotated)
missingok	Don't error if log absent
notifempty	Skip empty logs
create mode owner group	Create new empty log after rotate

Directive	Meaning
<code>copytruncate</code>	Copy then truncate in place (apps that never reopen)
<code>dateext / dateformat</code>	Date-based suffixes
<code>sharedscripts</code>	Run <code>postrotate</code> once for the whole block
<code>postrotate ... endscript</code>	Reload/signal app after rotate
<code>su user group</code>	Run as user when directories are restricted
<code>olddir</code>	Move rotated files elsewhere
<code>maxage days</code>	Remove rotated logs older than N days

Key Use Cases

1. Keep `/var/log` from filling root
2. Rotate custom application logs with reload hooks
3. Force rotation during an incident when a log explodes
4. Dry-run config after edits before production force

Examples with Explanations

Dry-run entire system config

```
sudo logrotate -d /etc/logrotate.conf
```

Shows which logs **would** rotate and why (or why not). No files changed. Always do this after editing drop-ins.

Force one drop-in (verbose)

```
sudo logrotate -f -v /etc/logrotate.d/rsyslog
# or your app:
sudo logrotate -f -v /etc/logrotate.d/myapp
```

Use when testing a new stanza or recovering from a multi-GB log that should have rotated.

Inspect state

```
sudo cat /var/lib/logrotate/status | head -50
sudo grep myapp /var/lib/logrotate/status
```

State tracks last rotation times; corrupted/stale state can skip or confuse schedules — know where it lives.

Example application drop-in

```
# /etc/logrotate.d/myapp
/var/log/myapp/*.log {
    daily
    rotate 14
    missingok
    notifempty
    compress
    delaycompress
    create 0640 myapp myapp
    sharedscripts
    postrotate
        systemctl reload myapp >/dev/null 2>&1 || true
    endscript
}
```

After install:

```
sudo logrotate -d /etc/logrotate.d/myapp
sudo logrotate -f -v /etc/logrotate.d/myapp
ls -la /var/log/myapp/
```

copytruncate when the app won't reopen

```
/var/log/legacy/app.log {
    size 100M
    rotate 7
    copytruncate
    missingok
    compress
    delaycompress
}
```

`copytruncate` avoids needing a signal, but can race and drop a few lines. Prefer apps that reopen on `SIGHUP`/`systemctl reload` with `create` + `postrotate`.

size-based emergency pattern

```
/var/log/myapp/debug.log {
    size 500M
    rotate 5
    compress
    missingok
}
```

```
    notifempty
    copytruncate
}
```

Helps when a debug flag left on would otherwise fill the disk overnight.

When does it run? (Ubuntu)

```
systemctl list-timers | grep -i logrotate
systemctl status logrotate.timer logrotate.service
# classic cron path may still exist on some releases:
ls /etc/cron.daily/logrotate
```

Timers can delay runs if the machine was off (`Persistent=` semantics vary).

Disk pressure triage with logs

```
df -h /var/log
sudo du -xh /var/log | sort -h | tail
sudo logrotate -f -v /etc/logrotate.conf
# journal is separate:
journalctl --disk-usage
sudo journalctl --vacuum-size=500M
```

Understanding Output

`-d` / `-v` lines explain decisions: “log does not need rotating”, “log needs rotating”, compression steps, and postrotate execution. Failures often come from permissions (`su` needed), missing directories, or postrotate scripts exiting non-zero.

Exit code non-zero should be treated as config or script failure — check unit logs:

```
journalctl -u logrotate.service -n 50
```

Notes & Pitfalls

- **Journald** `logrotate` — vacuum the journal separately.
- Prefer `reload/reopen` over `copytruncate` when the app supports it.
- After rotate, confirm the app still writes to the live path (not a deleted inode).
- `sharedscripts` + wildcards: `postrotate` runs once; without it, once per file.
- Permissions: log dirs owned by apps may need `su` in the stanza.
- Compressing huge logs can spike CPU/disk — `delaycompress` spreads pain.

- Don't rotate active database files or non-log data with logrotate recipes.
- Test with `-d` in CI or staging before shipping drop-ins.

Related Commands

- `logger` — write to syslog
- `journalctl` — systemd journal (vacuum, query)
- `systemctl reload` — reopen logs cleanly
- `df` / `du` — disk pressure
- `lsof` — deleted-but-open logs still holding space

Additional Resources

- `man logrotate`
- `man logrotate.conf`
- `/etc/logrotate.conf` and `/etc/logrotate.d/*` on the host

Hardware

Intro

Intro

Inventory and inspect physical devices: whole-system reports, PCI devices, DMI/SMBIOS data, and disk identity/timings. Use before opening tickets or swapping hardware.

Commands in this part

Command	Role
<code>lshw</code>	<code>lshw</code> (list hardware) probes the system and prints a hardware tree: CPU, memory, storage, network, firmware, and buses.
<code>lspci</code>	<code>lspci</code> lists PCI and PCI Express devices: NICs, GPUs, storage controllers, USB controllers, bridges, and more.
<code>dmidecode</code>	<code>dmidecode</code> dumps DMI/SMBIOS tables: manufacturer, product name, serial numbers, BIOS version, memory slot population,...
<code>hdparm</code>	<code>hdparm</code> gets and sets ATA/SATA disk parameters: identify data, power management, read-ahead, write-cache, and...

Suggested starting points

1. Broad inventory: `lshw`.
2. PCI devices: `lspci`.
3. Firmware/DMI: `dmidecode`.
4. Disk identity/timings: `hdparm` (SMART health is under Storage: `smartctl`).

Related parts

- System information — quick CPU/memory summaries
- Storage and filesystems — block devices and SMART

- Services and runtime — kernel modules via `modprobe`/`lsmod`

Continue with the individual command pages in this part.

lshw

Overview

`lshw` (list hardware) probes the system and prints a hardware tree: CPU, memory, storage, network, firmware, and buses. Run as **root** (or with capabilities) for full DMI/PCI detail; unprivileged runs omit sensitive or privileged data.

```
sudo apt install lshw
sudo dnf install lshw
```

Syntax

```
sudo lshw [options]
```

Common Options

Option	Description
<code>-short</code>	Compact table
<code>-class CLASS / -C</code>	Filter: network , disk , memory , cpu , system , display , storage , power , ...
<code>-businfo</code>	Bus addresses / logical names
<code>-json / -xml / -html</code>	Export formats
<code>-sanitize</code>	Hide serials and UUIDs for sharing
<code>-numeric</code>	Numeric IDs
<code>-quiet</code>	Less noise
<code>-enable / -disable tests</code>	Control probe tests (see man)

Key Use Cases

1. Inventory machines for asset management
2. Identify NIC driver/logical name mismatches
3. Confirm RAM/slots and disk models
4. Generate sanitized HTML/JSON for tickets

Examples with Explanations

Quick overview

```
sudo lshw -short
sudo lshw -class system -short
```

Network and disks

```
sudo lshw -class network -short
sudo lshw -C network -businfo
sudo lshw -class disk
sudo lshw -class storage
```

Memory and CPU

```
sudo lshw -class memory
sudo lshw -class cpu -short
```

Export

```
sudo lshw -json > /tmp/hw.json
sudo lshw -sanitize -html > /tmp/hw.html
sudo lshw -json | jq '.[].class' 2>/dev/null | sort -u
```

One-liner recipes

```
# MAC and product for NICs
sudo lshw -class network | egrep 'description|product|serial|logical name|capacity'

# Disk models
sudo lshw -class disk | egrep 'description|product|serial|size|logical name'

# Compare businfo to ip link
sudo lshw -C network -businfo; ip -br link
```

Notes & Pitfalls

- Without root, expect incomplete PCI/DMI data.
- **Virtual machines** report virtual hardware; use cloud metadata APIs for instance type.
- Probing can be slow on large systems; class filters help.
- Serial numbers are sensitive—use `-sanitize` before pasting into public issues.
- Prefer `lscpu`, `lsblk`, `ip link` for single-purpose questions; `lshw` is the wide inventory tool.

2026-relevant notes

- On servers, `dmidecode`, `lshw`, and `lscpu/lsblk` remain the classic trio; some fleets use `inxi`, `hwinfo`, or vendor out-of-band (iDRAC/iLO) for authoritative inventory.
- NVMe and virtual NICs rename often (`ens`, `enp`, `eth`)—trust `businfo` + `ip link` together.
- JSON export helps config-management facts when agent-based inventory isn't installed.

Related Commands

- `dmidecode` — SMBIOS/DMI tables
- `lspci` / `lsusb` — buses
- `lscpu` / `lsblk` / `lsmem` — focused views
- `ip link` — live interfaces
- `hwinfo` — alternate prober (SUSE heritage)

Additional Resources

- `man lshw`
- Upstream ezix.org `lshw` documentation

lspci

Overview

`lspci` lists **PCI and PCI Express** devices: NICs, GPUs, storage controllers, USB controllers, bridges, and more. Part of the `pciutils` package (standard on Ubuntu servers). First stop for “is the NIC visible?”, “which driver is bound?”, and collecting `vendor:device` IDs for driver or firmware work.

Syntax

```
lspci [options]
```

Common Options

Option	Description
<code>-v</code> / <code>-vv</code> / <code>-vvv</code>	Increasing verbosity (capabilities, BARs, ...)
<code>-k</code>	Kernel driver in use and modules that can bind
<code>-nn</code>	Numeric vendor:device (and class) codes
<code>-s slot</code>	Only this slot (<code>[[domain:]bus:]slot[.func]</code>)
<code>-d vendor:device</code>	Filter by IDs (8086:, 10de:, ...)
<code>-t</code>	Tree of buses/bridges
<code>-mm</code>	Machine-readable key-value
<code>-D</code>	Always show domain numbers
<code>-n</code>	Numeric IDs only (no name resolution)
<code>-x</code> / <code>-xxx</code>	Hex config space dump (needs root for full)

Key Use Cases

1. Confirm a device is enumerated after install/passthrough
2. See which kernel driver owns a NIC/GPU/storage HBA
3. Collect `vendor:device` for DKMS/out-of-tree modules
4. Compare before/after firmware or hardware changes

Examples with Explanations

Everyday list

```
lspci
lspci -nn
```

Human names plus, with `-nn`, stable [vvvv:dddd] IDs for docs and `modprobe` quirks.

Drivers and modules

```
lspci -k
lspci -nnk
```

Kernel driver in use is the bound driver; Kernel modules lists candidates. Empty driver often means missing firmware, blacklisted module, or unbound VFIO for passthrough.

Network / GPU filters

```
lspci | grep -i ethernet
lspci | grep -i network
lspci | grep -i nvidia
lspci | grep -i 'non-volatile\|nvme\|raid\|scsi'
lspci -nnk | grep -A3 -i ethernet
```

One device, deep detail

```
lspci -s 00:1f.2 -vv
sudo lspci -s 00:1f.2 -vvv
```

Slot address from the first column of `lspci`. Root unlocks more config space in deep dumps.

Tree view

```
lspci -t
lspci -tv
```

Shows how devices hang off bridges — useful for bandwidth/topology discussions and multi-GPU layouts.

Machine-readable

```
lspci -mmnn  
lspci -mm | head
```

Easier for scripts than scraping the pretty format.

ID database refresh

```
sudo update-pciids  
lspci -nn | head
```

Refreshes `/usr/share/misc/pci.ids` so new devices resolve to names (IDs still work offline).

Correlate with sysfs / ethtool

```
lspci -nnk | grep -A3 -i ethernet  
# then for a netdev:  
ethtool -i eth0  
ls -l /sys/class/net/eth0/device/driver
```

`lspci` shows PCI binding; `ethtool -i` shows the netdev's driver/firmware strings.

Understanding Output

Default line shape:

```
00:1f.2 SATA controller: Intel Corporation ... [8086:.... ] (rev ..)  
  
                                human name          vendor:device  
bus:slot.func + class
```

Piece	Role
Domain/BDF	Address for <code>-s</code> and sysfs paths
Class	Ethernet, VGA, NVMe, bridge, ...
Vendor/device IDs	Stable identity for drivers
<code>-k</code> lines	Bound driver vs available modules

Sysfs path pattern: `/sys/bus/pci/devices/0000:bus:slot.func/`.

Notes & Pitfalls

- Some extended reads need **root** (**-vvv**, full **-xxx**).
- Names depend on **pci.ids**; unknown devices still show IDs with **-nn**.
- SR-IOV VFs appear as extra devices — don't confuse PF vs VF when binding drivers.
- GPU “not found” can be power/BIOS/resizability (BAR) issues — **dmesg** matters.
- Hotplug/cloud passthrough: re-run **lspci** after attach; guest may need reboot for some devices.
- **update-pciids** needs network; air-gapped hosts keep stale names only.

Related Commands

- **lsusb** — USB bus inventory
- **lshw -class network -class display** — higher-level inventory
- **ethtool** — NIC driver/link settings
- **modinfo** / **lsmod** / **modprobe** — module details and load
- **dmesg** / **journalctl -b** — probe/firmware errors
- **setpci** — low-level PCI config (expert; easy to break systems)

Additional Resources

- **man lspci**
- **man update-pciids**
- **man pci.ids**

dmidecode

Overview

`dmidecode` dumps **DMI/SMBIOS** tables: manufacturer, product name, serial numbers, BIOS version, memory slot population, processor strings, chassis type, and more. It is the go-to CLI for “what hardware is this?” on bare metal without opening a ticket to the DC. Requires **root**. VMs expose synthetic or partial DMI data.

Syntax

```
sudo dmidecode [options]
```

Common Options

Option	Description
<code>-t type</code>	Restrict by type name or number: <code>system</code> , <code>bios</code> , <code>baseboard</code> , <code>memory</code> , <code>processor</code> , <code>chassis</code> , <code>slot</code> , ...
<code>-s keyword</code>	Print a single string (<code>system-serial-number</code> , <code>system-product-name</code> , <code>bios-version</code> , ...)
<code>-u</code>	Dump raw formatted entries
<code>-q</code>	Quieter (less decoration)
<code>-d file</code>	Read from dump file instead of <code>/sys/firmware/dmi/tables</code>
<code>--oem-string N</code>	OEM string index

Common `-t` names: `bios`, `system`, `baseboard`, `chassis`, `processor`, `memory`, `cache`, `connector`, `slot`.

Key Use Cases

1. Inventory serial / product / UUID for asset tracking
2. BIOS/UEFI version before firmware upgrades
3. Memory slot map (which DIMM empty/populated)
4. Confirm bare-metal vs obvious VM DMI strings

Examples with Explanations

System identity

```
sudo dmidecode -t system
sudo dmidecode -s system-manufacturer
sudo dmidecode -s system-product-name
sudo dmidecode -s system-serial-number
sudo dmidecode -s system-uuid
```

Serial and UUID are sensitive — scrub before pasting into public tickets.

BIOS / firmware

```
sudo dmidecode -t bios
sudo dmidecode -s bios-version
sudo dmidecode -s bios-release-date
sudo dmidecode -s bios-vendor
```

Capture before and after firmware updates.

Memory population

```
sudo dmidecode -t memory
sudo dmidecode -t memory | grep -E 'Size:|Locator:|Bank Locator:|Speed:|Manufacturer:|Part N
```

Shows per-slot size/speed; empty slots often report No Module Installed / size No Module Installed.

Processor summary

```
sudo dmidecode -t processor
sudo dmidecode -t processor | grep -E 'Version:|Core Count:|Core Enabled:|Thread Count:|Max
```

Complement with `lscpu` for the kernel's view of topology and flags.

Baseboard and chassis

```
sudo dmidecode -t baseboard
sudo dmidecode -t chassis
sudo dmidecode -s baseboard-serial-number
sudo dmidecode -s chassis-type
```

Useful for blade/rack asset fields and chassis form factor.

Inventory one-liners for scripts

```
sudo dmidecode -s system-manufacturer
sudo dmidecode -s system-product-name
sudo dmidecode -s system-serial-number
sudo dmidecode -s bios-version
```

`-s` fails closed if the keyword is invalid — check `man dmidecode` for the keyword list.

Compare with sysfs (no full dump)

```
# many fields also appear here (root not always required for all):
cat /sys/class/dmi/id/product_name
cat /sys/class/dmi/id/sys_vendor
cat /sys/class/dmi/id/product_serial # may be root-only / permission-denied
```

`sysfs` is handy for a few fields; `dmidecode` is better for structured tables (memory slots).

Understanding Output

Output is grouped into **handles** (DMI record instances). Each handle has a type (e.g. Memory Device) and key/value fields. Multiple handles of the same type are normal (one per DIMM, per CPU socket, ...).

Area	What operators extract
System	Vendor, model, serial, UUID
BIOS	Version, date, vendor
Memory Device	Locator, size, speed, part number
Processor	Version string, core/thread counts

Garbage or Not Specified / To be filled by O.E.M. means the vendor left SMBIOS incomplete — common on some whiteboxes and VMs.

Notes & Pitfalls

- **Root required** for full access; without it you get little or nothing.
- VMs (KVM, VMware, cloud) synthesize DMI — serials may be instance IDs or placeholders.
- Serial numbers are **PII/asset-sensitive** — redact in chat logs.
- DMI can be wrong if vendors mess up tables; cross-check labels and BMC/iDRAC/iLO when stakes are high.

- Not a substitute for `lshw` PCI detail or `lspci` driver binding.
- `system-uuid` may be used by software licensing — changing it (rare/firmware) has consequences.

Related Commands

- `lshw` — broader hardware tree (also often needs root)
- `lscpu` — CPU topology/flags from kernel
- `lspci` / `lsusb` — buses and peripherals
- `cat /sys/class/dmi/id/*` — quick sysfs fields
- `dmesg` — firmware/kernel hardware messages

Additional Resources

- `man dmidecode`
- SMBIOS field quality varies by OEM; trust but verify against physical tags for RMA.

hdparm

Overview

`hdparm` gets and sets **ATA/SATA disk** parameters: identify data, power management, read-ahead, write-cache, and controversial performance timings. Useful for bare-metal disk troubleshooting. Wrong settings can risk **data loss** — especially write-cache and dangerous benchmark flags.

Prefer vendor tools / `smartmontools` for health; use `hdparm` carefully.

Syntax

```
hdparm [options] device
```

Device is typically `/dev/sda` (whole disk), not a partition.

Common Options

Option	Description
<code>-I</code>	Detailed identity information
<code>-i</code>	Kernel's identify summary
<code>-C</code>	Power mode status
<code>-M</code>	AAM acoustic (legacy)
<code>-B</code>	APM level
<code>-S</code>	Standby timeout
<code>-W</code>	Write-cache on/off query/set
<code>-a / -A</code>	Read-ahead
<code>-t / -T</code>	Timing buffered disk / cache reads (benchmark load)
<code>--security-*</code>	ATA security (dangerous)
<code>-y / -Y</code>	Standby / sleep now
<code>-z</code>	Reread partition table

Examples with Explanations

Identify

```
sudo hdparm -I /dev/sda
sudo hdparm -i /dev/sda
sudo smartctl -a /dev/sda
```

Power status

```
sudo hdparm -C /dev/sda
```

Read-only info queries (safer)

```
sudo hdparm -W /dev/sda
sudo hdparm -a /dev/sda
```

Benchmarks (I/O heavy; maintenance window)

```
sudo hdparm -tT /dev/sda
```

Interprets sequential throughput roughly; not a substitute for `fio`.

Write cache (know the risk)

```
sudo hdparm -W /dev/sda
# enabling write cache without barriers/UPS can lose data on power loss
```

Prefer filesystem/mount and device defaults unless you know the storage stack.

Reread partitions

```
sudo hdparm -z /dev/sda
# often partprobe / blockdev --rereadpt also used
```

Notes / Pitfalls

- **Never** experiment with `--security-erase` / passwords on disks you care about without full understanding.
- USB bridges and NVMe devices may ignore or poorly implement ATA commands — NVMe has `nvme` CLI.
- Cloud block volumes often virtualize away `hdparm` usefulness.
- Timing tests stress disks — avoid on production peaks.
- Requires root.

2026-relevant notes

- NVMe dominates SSDs: learn `nvme list`, `nvme smart-log` alongside `hdparm`.
- For spinning rust power policies on laptops, distro power tools may be safer than manual `-B/-S`.
- Always have backups before storage experiments.

Related Commands

- `smartctl` — SMART health
- `nvme` — NVMe admin
- `lsblk` / `fdisk` / `blkid` — topology
- `fio` — proper benchmarks
- `blockdev` — low-level block ioctls

Additional Resources

- `man hdparm` (read warnings)

Storage and Filesystems

Intro

Intro

Block devices, partitions, filesystems, mounts, LVM, encryption, integrity checks, and space usage. Destructive tools (`mkfs`, `dd`, `wipefs`, `cryptsetup luksFormat`) always need identity double-checks.

Commands in this part

Command	Role
<code>lsblk</code>	<code>lsblk</code> lists block devices (disks, partitions, LVM logical volumes, loop devices, ROM) as a tree.
<code>blkid</code>	<code>blkid</code> probes block devices for filesystem, RAID, and crypto signatures and prints tags such as TYPE, UUID, and LABEL.
<code>findmnt</code>	<code>findmnt</code> lists or searches mounted filesystems in tree or table form.
<code>fdisk</code>	<code>fdisk</code> creates and edits DOS/MBR and GPT partition tables on block devices.
<code>mkfs</code>	<code>mkfs</code> builds a filesystem on a block device (partition, LV, loop file).
<code>fsck</code>	<code>fsck</code> checks and optionally repairs filesystems.
<code>mount</code>	<code>mount</code> attaches a filesystem (block device, network share, bind path, tmpfs, ...) onto a directory in the single Linux...
<code>umount</code>	<code>umount</code> detaches a mounted filesystem from the directory tree.
<code>dd</code>	<code>dd</code> copies and converts data at the block level.
<code>sync</code>	<code>sync</code> flushes filesystem buffers: dirty page cache and pending metadata are written to storage.
<code>losetup</code>	<code>losetup</code> associates regular files with loop devices (<code>/dev/loopN</code>) so the kernel can treat an image file like a block...

Command	Role
<code>duf</code>	<code>duf</code> is a modern, colorful disk usage / free space viewer — a friendlier alternative to <code>df -h</code> .
<code>dust</code>	<code>dust</code> is a modern <code>du</code> alternative that shows disk usage as a readable tree/bars, helping you find large directories...
<code>parted</code>	<code>parted</code> creates and modifies partition tables (GPT/MSDOS) on block devices.
<code>wipefs</code>	<code>wipefs</code> probes and can erase filesystem, RAID, or partition-table signatures from a device without zeroing the entire...
<code>smartctl</code>	<code>smartctl</code> (from <code>smartmontools</code>) queries disk S.M.A.R.T.
<code>pvs</code>	<code>pvs</code> reports LVM physical volumes (PVs) — disks or partitions initialized with <code>pvcreate</code> and used by volume groups.
<code>vgs</code>	<code>vgs</code> reports LVM volume groups (VGs): how physical volumes are pooled and how much space is free for logical volumes.
<code>lvs</code>	<code>lvs</code> reports LVM logical volumes (LVs) — the block devices you usually format and mount (<code>/dev/VG/LV</code> or...
<code>cryptsetup</code>	<code>cryptsetup</code> manages LUKS (and other) disk encryption: format, open, close, and resize encrypted block devices.
<code>resize2fs</code>	<code>resize2fs</code> grows or shrinks <code>ext2/ext3/ext4</code> filesystems.
<code>tune2fs</code>	<code>tune2fs</code> adjusts tunable parameters on <code>ext2/ext3/ext4</code> filesystems: labels, UUID, mount-count checks, reserved blocks,...
<code>ncdu</code>	<code>ncdu</code> (NCurses Disk Usage) is an interactive <code>du</code> for finding what consumes space.

Suggested starting points

1. See devices: `lsblk`, `blkid`, `findmnt`.

2. Partition: `parted/fdisk`, then `mkfs`, `mount/umount`.
3. Check/repair: `fsck` (unmounted); grow/tune ext*: `resize2fs`, `tune2fs`.
4. LVM trio: `pvs`, `vgs`, `lvs`.
5. Encryption: `cryptsetup`.
6. Space: `duf/dust/ncdu`; health: `smartctl`.

Related parts

- Files and paths — `df/du` at the path layer
- Hardware — disk controllers and DMI
- Security — labels on mounted data

Continue with the individual command pages in this part.

lsblk

Overview

`lsblk` lists block devices (disks, partitions, LVM logical volumes, loop devices, ROM) as a tree. First stop when you plug in a disk, write `fstab` entries, or answer “what is `/dev/sdb1`?”. It shows topology and filesystem metadata; it does **not** report free space inside filesystems — use `df` / `du` for capacity.

Syntax

```
lsblk [options] [device...]
```

Common Options

Option	Description
<code>-f</code>	Filesystem view: FSTYPE, LABEL, UUID, FSAVAIL, FSUSE%, mountpoints
<code>-o COLS</code>	Custom column list
<code>-p</code>	Full device paths (<code>/dev/sda</code> not <code>sda</code>)
<code>-a</code>	Include empty devices
<code>-d</code>	Devices only (no partition children)
<code>-e 7</code>	Exclude major number 7 (loop) — hide snap noise
<code>-J</code> / <code>-0</code>	JSON / all known columns
<code>-t</code>	Topology (alignment, scheduler, discard)
<code>-S</code>	SCSI devices only
<code>-n</code>	No header (script-friendly with <code>-o</code>)
<code>-r</code>	Raw (no tree indentation)

Useful columns: NAME, SIZE, TYPE, FSTYPE, LABEL, UUID, MOUNTPOINTS, MODEL, TRAN, ROTA, SERIAL, PARTUUID.

Key Use Cases

1. Inventory disks after attach (USB, cloud volume, NVMe)
2. Collect UUID/LABEL for durable fstab lines
3. Distinguish SSD/NVMe vs spinning (ROTA, TRAN)
4. Map LVM/mapper children under parent disks
5. Feed scripts with JSON or selected columns

Examples with Explanations

Everyday inventory

```
lsblk
lsblk -f
lsblk -p
```

Default tree shows NAME, MAJ:MIN, RM, SIZE, RO, TYPE, MOUNTPOINTS. `-f` adds FSTYPE/LABEL/UUID — the view you want before mount or fstab work. `-p` prints full paths for copy-paste into commands.

UUID-ready view for fstab

```
lsblk -o NAME,SIZE,FSTYPE,LABEL,UUID,MOUNTPOINTS
lsblk -o NAME,PARTUUID,UUID,FSTYPE,MOUNTPOINTS -p
```

Prefer UUID= (or PARTUUID= for partitions without a FS yet) over `/dev/sdX` names that can reorder on reboot.

Hide loop noise (snaps)

```
lsblk -e 7
lsblk -e 7 -f
# or filter: lsblk | grep -v loop
```

Ubuntu systems with many snaps flood the tree with loop devices. Major 7 is loop; exclude it for human scanning.

JSON for scripts

```
lsblk -J -o NAME,SIZE,TYPE,FSTYPE,MOUNTPOINTS | jq .  
lsblk -J -o NAME,SIZE,TYPE,MOUNTPOINTS | jq -r '.blockdevices[].name'
```

Stable structure for automation; combine with `jq` rather than parsing the pretty tree.

Topology / rotational / transport

```
lsblk -d -o NAME,SIZE,ROTA,TYPE,TRAN,MODEL,SERIAL  
# ROTA=0 often SSD/NVMe; TRAN=nvme|sata|usb|sas
```

`-d` limits to whole disks. Use this when capacity planning or placing I/O-heavy workloads (prefer `TRAN=nvme` over USB).

One device subtree

```
lsblk -f /dev/nvme0n1  
lsblk -f /dev/sdb
```

Focus on a single parent and its partitions/holders — less noise on multi-disk hosts.

Align with blkid and mounts

```
lsblk -f  
sudo blkid  
findmnt -D
```

Cross-check: `lsblk` topology, `blkid` superblock tags, `findmnt` what is actually mounted.

Empty or new disk check

```
lsblk -f /dev/sdb  
# no FSTYPE / no children → raw or unpartitioned; confirm size matches expectation
```

Before `fdisk/mkfs`, verify size and that nothing is already mounted on that path.

Understanding Output

Field	Meaning
Tree indent	Parent disk → partitions → LVM/dm children
TYPE	disk, part, lvm, crypt, loop, rom, ...
FSTYPE	Superblock type (ext4, xfs, vfat, crypto_LUKS, ...); empty raw/unformatted
UUID / LABEL	Filesystem identity for fstab
MOUNTPOINTS	Where mounted (multiple possible with binds on recent util-linux)
ROTA	1 rotational disk, 0 usually SSD/NVMe
TRAN	Transport: nvme, sata, usb, ...
RM	Removable flag

Kernel device names (`sda` vs `sdb`, `nvme0n1p1`) are **not** stable across boots or controller probe order.

Notes & Pitfalls

- Kernel names change; always pin mounts with UUID/LABEL/PARTUUID.
- `lsblk` is not free-space accounting — use `df -h` / `findmnt -D` for capacity.
- Some details need root; listing usually works unprivileged.
- Multipath hosts may show both `/dev/sdX` and `/dev/mapper/mpath*` — mount the multipath node.
- Cloud “detach/attach” can renumber disks; re-run `lsblk` after attach before scripting.
- RAID/LVM: format/mount the logical device, not a single member disk.

Related Commands

- `blkid` — probe UUID/LABEL/TYPE from superblocks
- `findmnt` — mounts as tree/table/JSON
- `df` / `duf` — capacity of mounted filesystems
- `fdisk` / `parted` — partition tables
- `lvs` / `vgs` / `pvs` — LVM detail
- `ls -l /dev/disk/by-uuid` — udev stable symlinks

Additional Resources

- `man lsblk`
- `man 8 lsblk` column list (`--help` also lists columns)

blkid

Overview

blkid probes block devices for filesystem, RAID, and crypto signatures and prints tags such as **TYPE**, **UUID**, and **LABEL**. Use it when building durable `/etc/fstab` lines, verifying a format succeeded, or resolving `UUID=` / `LABEL=` to a device path. Complements `lsblk -f` (topology) with superblock-focused output.

Syntax

```
sudo blkid [options] [device...]
```

Without a device, lists known devices (may use cache; root improves completeness).

Common Options

Option	Description
<code>-p</code>	Low-level probe (bypass some cache behavior)
<code>-s TAG</code>	Show only tag(s): <code>UUID</code> , <code>LABEL</code> , <code>TYPE</code> , <code>PARTUUID</code> , ...
<code>-o value\ export\ list\ udev\ full</code>	Output format
<code>-U UUID</code>	Look up device path by filesystem UUID
<code>-L LABEL</code>	Look up device path by LABEL
<code>-c /dev/null</code>	Ignore cache file (force fresh probe)

Key Use Cases

1. Copy UUID into fstab after `mkfs`
2. Resolve `UUID=` or `LABEL=` to `/dev/...` path
3. Confirm filesystem type before `mount/fsck`
4. Script-friendly single-value extraction (`-o value`)

Examples with Explanations

List all probed devices

```
sudo blkid
sudo blkid -c /dev/null
```

Shows TYPE/UUID/LABEL (and more) per device. `-c /dev/null` avoids stale cache after recent formats.

One device

```
sudo blkid /dev/sdb1
sudo blkid -p /dev/sdb1
```

Export form (shell-friendly)

```
sudo blkid -o export /dev/sdb1
# DEVNAME=/dev/sdb1
# UUID=...
# TYPE=ext4
```

Easy to `eval` carefully or parse line-by-line in scripts.

UUID only (fstab snippets)

```
sudo blkid -s UUID -o value /dev/sdb1
UUID=$(sudo blkid -s UUID -o value /dev/sdb1)
echo "UUID=$UUID /mnt/data ext4 defaults 0 2"
```

Prefer this over hard-coding `/dev/sdX`.

Lookup by UUID or LABEL

```
sudo blkid -U 123e4567-e89b-12d3-a456-426614174000
sudo blkid -L data
ls -l /dev/disk/by-uuid/
ls -l /dev/disk/by-label/
```

Returns the device node currently bound to that identity. `udev` symlinks under `/dev/disk/by-*` are the other stable interface.

Compare with lsblk

```
lsblk -f
sudo blkid
```

`lsblk -f` is tree-oriented; `blkid` is probe-oriented. Use both when something disagrees (cache, multipath aliases).

After mkfs verification

```
sudo mkfs.ext4 -L data /dev/sdb1
sudo blkid -c /dev/null /dev/sdb1
lsblk -f /dev/sdb1
```

Confirm new UUID/LABEL before writing `fstab`.

udev property style

```
sudo blkid -o udev /dev/sdb1
# ID_FS_UUID=...
# ID_FS_TYPE=...
```

Matches tags `udev` exposes; useful when correlating with `udevadm info`.

Understanding Output

Typical line:

```
/dev/sdb1: UUID="..." BLOCK_SIZE="4096" TYPE="ext4" PARTUUID="..."
```

Tag	Role
TYPE	Filesystem/crypto/RAID signature
UUID	Filesystem UUID (<code>fstab UUID=</code>)
LABEL	Human label (<code>LABEL=</code>)
PARTUUID	GPT partition UUID (stable even before FS)
PARTLABEL	GPT partition name

No output often means no recognized signature (raw, wiped, or inaccessible without root).

Notes & Pitfalls

- Root is often required for reliable probing of all devices.
- Cache can lag immediately after format — use `-c /dev/null` or `-p` when suspicious.
- Multipath/LVM: same UUID may appear under several paths; mount via mapper/multipath node consistently.
- LABEL collisions: two disks with LABEL=data make LABEL= fstab ambiguous — prefer UUID.
- LUKS: outer device shows `crypto_LUKS`; filesystem UUID appears on the opened mapper device.
- Swap shows TYPE="swap" with its own UUID (`/etc/crypttab` / fstab swap lines).

Related Commands

- `lsblk -f` — topology + FS columns
- `findmnt` — active mounts
- `udevadm info -q property -n /dev/sdb1` — udev properties
- `ls /dev/disk/by-uuid` — stable symlinks
- `wipefs -n` — list signatures without blkid cache

Additional Resources

- `man blkid`
- `man fstab`

findmnt

Overview

`findmnt` lists or searches mounted filesystems in tree or table form. It is clearer than parsing `mount` output and friendlier for scripts (`-J`, `-n`, `-o`, `-r`). First tool for “what is mounted where?”, `fstab` verification, and busy-mount debugging before `umount`.

Syntax

```
findmnt [options] [device|mountpoint]
```

Common Options

Option	Description
(none)	Tree of all mounts
<code>-t types</code>	Filter by FS types (<code>ext4,xfs,nfs</code>)
<code>-S source</code>	Match source device/path
<code>-T path</code>	Filesystem that contains this path
<code>-D</code>	df-like size columns
<code>-o list</code>	Select columns
<code>-n</code>	No header
<code>-r</code>	Raw (no tree art)
<code>-J</code>	JSON
<code>-f</code>	First match only
<code>-R</code>	Sub-tree of a mountpoint
<code>--verify</code>	Check <code>/etc/fstab</code> against reality (util-linux)
<code>-A / -l</code>	All / list format

Useful columns: `TARGET`, `SOURCE`, `FSTYPE`, `OPTIONS`, `UUID`, `LABEL`, `SIZE`, `USE%`.

Key Use Cases

1. Human-readable mount inventory (better than raw `mount`)
2. Which mount owns a path (`-T`)
3. df-like capacity with mount context (`-D`)
4. Scripted JSON/table for monitoring
5. Verify fstab before reboot (`--verify`)

Examples with Explanations

Full tree

```
findmnt
findmnt -t ext4,xfs,btrfs,nfs
```

Tree shows mount hierarchy (binds and API filesystems included). Filter types to cut tmpfs/proc noise when scanning “real” disks.

df-like sizes

```
findmnt -D
findmnt -D -t ext4,xfs
```

Combines mount table with used/available — often nicer than juggling `df` + `mount`.

What holds this path?

```
findmnt -T /var/log
findmnt -T .
findmnt -T /var/lib/docker
```

Critical with bind mounts and nested volumes: `df /var/lib/docker` can surprise you; `-T` shows the owning mount.

By source or target

```
findmnt /
findmnt -S /dev/sdb1
findmnt /mnt/data
findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS /mnt/data
```

Script-friendly

```
findmnt -n -o SOURCE /
findmnt -nro TARGET,FSTYPE
findmnt -J | jq -r '.filesystems[].target'
```

`-n` drops headers; `-r` raw fields; JSON for structured tools.

Busy mount debugging

```
findmnt /mnt/data
findmnt -R /mnt/data
sudo lsof +f -- /mnt/data | head
sudo fuser -vm /mnt/data
```

See exact mount options and nested mounts before `umount`.

Verify fstab

```
findmnt --verify
findmnt --verify --verbose
```

Catches missing devices, bad options, or stale UUIDs **before** the next boot fails.

NFS / bind focus

```
findmnt -t nfs,nfs4
findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS -t nfs,nfs4
findmnt | grep bind    # binds often show same source path twice
```

Understanding Output

Column	Meaning
TARGET	Mountpoint
SOURCE	Device, UUID path, or bind source
FSTYPE	Filesystem type
OPTIONS	<code>rw</code> , <code>noexec</code> , <code>nosuid</code> , NFS opts, ...
Size cols (-D)	Capacity like <code>df</code>

Tree indentation reflects mount nesting (e.g. bind under `/mnt`). Pseudo filesystems (`proc`, `sysfs`, `cgroup2`) dominate unfiltered views on modern systemd systems.

Notes & Pitfalls

- Unfiltered output is noisy — filter with `-t` or look at specific targets.
- Namespace matters: inside containers/`unshare` you see that mount NS only.
- `findmnt -T` follows the mount covering a path, not “directory exists”.
- Options string is long; use `-o` to extract what you need.
- Prefer `findmnt` over scraping `/proc/mounts` in new scripts (still OK to read `/proc/self/mountinfo` for deep debugging).

Related Commands

- `mount / umount` — attach/detach
- `lsblk -f / blkid` — device identity
- `df -h` — classic capacity
- `lsof / fuser` — busy mounts
- `systemctl status *.mount` — systemd mount units

Additional Resources

- `man findmnt`
- `man mount`
- `man fstab`

fdisk

Overview

`fdisk` creates and edits DOS/MBR and GPT partition tables on block devices. Interactive by default; list-only with `-l`. For scripted GPT layouts many operators prefer `sfdisk` or `parted`; for pure GPT recovery, `gdisk` is another common tool. After partitioning you still need `mkfs` (or LVM/crypt setup) before data use.

Syntax

```
sudo fdisk [-l] [device...]  
sudo fdisk device          # interactive editor
```

Common Options

Option	Description
<code>-l</code>	List partition tables (no write)
<code>-x</code>	Extra detail when listing
<code>-w always\ never</code>	Wipe signatures when writing (newer util-linux)
<code>-t dos\ gpt</code>	Restrict label type (context-dependent)

Interactive keys (common):

Key	Action
<code>m</code>	Help
<code>p</code>	Print table
<code>g</code>	New empty GPT
<code>o</code>	New empty DOS/MBR
<code>n</code>	New partition
<code>d</code>	Delete partition
<code>t</code>	Change type (Linux filesystem, LVM, EFI, ...)
<code>w</code>	Write table to disk and exit
<code>q</code>	Quit without saving

Safety

Writing a partition table can destroy access to existing data. Treat `w` as irreversible without backups.

1. Identify the **correct** disk: `lsblk -f`, `lsblk -o NAME,SIZE,MODEL,SERIAL,TRAN`, `sudo fdisk -l`.
2. Never guess `sdb` vs `sdc` on multi-disk hosts — match size/model/serial.
3. Do not rewrite tables on mounted system disks without a recovery plan and offline/maintenance window.
4. Prefer working on **data** disks first; practice on disposable VMs.
5. After write, re-read the table (`partprobe` / reboot if needed) before `mkfs`.

Key Use Cases

1. List all disks and partitions before storage work
2. Create GPT layout on a new data disk
3. Add a partition for LVM or a second filesystem
4. Dump/restore layouts via `sfdisk` for clones

Examples with Explanations

List all disks (read-only)

```
sudo fdisk -l
sudo fdisk -l /dev/sdb
sudo fdisk -l /dev/nvme0n1
```

Safe first step. Confirms capacity, label type (Disklabel type: gpt/dos), and existing partitions.

Confirm target before editing

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,MOUNTPOINTS,MODEL,SERIAL,TRAN
lsblk -f /dev/sdb
# only proceed if size/model match the disk you intend to wipe
```

Wrong-device mistakes are the most common disaster with `fdisk`/`mkfs`.

Interactive GPT data disk (sketch)

```
sudo fdisk /dev/sdb
# g          → new GPT
# n          → new partition (accept defaults for whole disk, or +200G etc.)
# t          → type if needed (Linux filesystem is typical; LVM has its own type)
# p          → review
# w          → write (point of no return for the old table)
```

Quit with `q` instead of `w` if anything looks wrong. Partition numbers and paths: `/dev/sdb1`, NVMe uses `/dev/nvme0n1p1`.

Kernel re-read after write

```
sudo partprobe /dev/sdb
# or: sudo blockdev --rereadpt /dev/sdb
lsblk /dev/sdb
cat /proc/partitions | grep sdb
```

If nodes do not appear, a reboot or removing leftover holders may be required.

Next steps after partitioning

```
sudo mkfs.ext4 -L data /dev/sdb1
# or: sudo pvcreate /dev/sdb1 && ... LVM ...
lsblk -f /dev/sdb
sudo blkid /dev/sdb1
```

`fdisk` only lays out partitions; filesystems and `fstab` are separate.

Scripted dump / restore (`sfdisk`)

```
sudo sfdisk -d /dev/sdb > sdb.layout
# restore to same-size disk (DESTRUCTIVE - verify target):
# sudo sfdisk /dev/sdb < sdb.layout
```

Use for clone pipelines; still verify device identity before restore.

EFI system partition type (boot disk work)

```
# In fdisk: type often "EFI System" (GPT). Wrong type can break boot loaders.  
# Prefer known-good installers for OS disks; manual ESP edits need care.
```

OS boot disks deserve extra caution — prefer installer tooling unless you know the boot chain.

Understanding Output

`fdisk -l` shows device path, size, sector size, disk identifier, and a table of partitions (number, start/end sectors, size, type). Modern util-linux `fdisk` aligns partitions automatically for 4K devices. GPT entries have PARTUUIDs visible via `lsblk -o NAME,PARTUUID` or `blkid`.

Notes & Pitfalls

- Device naming: SATA/SCSI `/dev/sdX`, NVMe `/dev/nvmeXnY` + `pN` partitions, virtio often `/dev/vdX`.
- Alignment is usually automatic; avoid manual cylinder math on modern disks.
- Changing a live root partition table is high risk — use rescue media.
- Residual filesystem signatures: wipe carefully (`wipefs`) only when intentional.
- Cloud images may already be GPT with growable partitions — don't blindly recreate.
- After type change to LVM/RAID, tools expect the matching stack (`pvcreate`, `mdadm`, ...).

Related Commands

- `parted` / `gdisk` — alternative partition editors
- `sfdisk` — scriptable dump/restore
- `mkfs` — create filesystem on a partition
- `lsblk` / `blkid` — verify layout and UUIDs
- `partprobe` / `blockdev --rereadpt` — refresh kernel view
- `wipefs` — inspect/clear signatures (destructive)

Additional Resources

- `man fdisk`
- `man sfdisk`
- `man parted`

mkfs

Overview

`mkfs` builds a filesystem on a block device (partition, LV, loop file). On Ubuntu you usually call the type-specific tools: `mkfs.ext4`, `mkfs.xfs`, `mkfs.vfat`, `mkfs.btrfs`. **This destroys existing data** on the target. Pair with `lsblk`/`blkid` for identity, `fdisk` for partitions, and `mount`/`fstab` for use.

Syntax

```
sudo mkfs -t type [fs-options] device
sudo mkfs.ext4 [options] device
sudo mkfs.xfs [options] device
sudo mkfs.vfat [options] device
```

Common Options

Options vary by filesystem. Operators commonly use:

Tool / option	Description
<code>mkfs.ext4 -L label</code>	Volume label
<code>mkfs.ext4 -m percent</code>	Reserved blocks for root (default ~5%)
<code>mkfs.ext4 -N inodes</code>	Inode count (rare; plan for many tiny files)
<code>mkfs.xfs -L label</code>	Label
<code>mkfs.xfs -f</code>	Force overwrite existing FS
<code>mkfs.vfat -F 32 -n NAME</code>	FAT32 + short volume name
<code>mkfs -t type</code>	Generic front-end

Swap is **not** `mkfs`: use `mkswap`.

Safety

1. Confirm device with `lsblk -o NAME,SIZE,MODEL,SERIAL,FSTYPE,MOUNTPOINTS` and `sudo blkid`.
2. Ensure the target is **unmounted** and not in use (`findmnt`, `lsof`, `fuser`).
3. Never `mkfs` the wrong disk on multi-disk or cloud hosts — match size/serial.
4. Prefer operating on `/dev/disk/by-id/...` or `by-uuid` only **after** you know the new FS UUID (identity changes on format).
5. Have backups; there is no undo.

Key Use Cases

1. Format a new data partition after `fdisk`
2. Format an LVM logical volume
3. Prepare USB media (vfat/exfat) for interchange
4. Rebuild a filesystem after intentional wipe (with backups)

Examples with Explanations

Verify target, then ext4 data disk

```
lsblk -f
lsblk -o NAME,SIZE,MODEL,SERIAL,MOUNTPOINTS /dev/sdb
sudo findmnt /dev/sdb1 # should be empty / not mounted
sudo mkfs.ext4 -L data /dev/sdb1
sudo blkid /dev/sdb1
sudo mkdir -p /mnt/data
sudo mount /dev/sdb1 /mnt/data
df -h /mnt/data
```

Label `-L data` shows up in `lsblk -f` and can be used as `LABEL=data` in `fstab` (UUID is still more unique).

XFS data volume

```
sudo mkfs.xfs -f -L data /dev/sdb1
```

`-f` forces if a filesystem signature already exists. XFS is common for large sequential workloads; repair path is `xfs_repair`, not classic `fsck` semantics.

FAT32 for USB interoperability

```
sudo mkfs.vfat -F 32 -n KEY /dev/sdc1
```

Good for firmware, Windows interchange, EFI-related media. Volume name length limits apply (-n).

Reduce ext* reserved space on large data disks

```
sudo mkfs.ext4 -m 0.5 -L bulk /dev/sdb1
# or later: sudo tune2fs -m 1 /dev/sdb1
```

Default ~5% reserved for root wastes terabytes on multi-TB bulk disks. Root filesystem often keeps a higher reserve on purpose.

LVM logical volume

```
# after lvcreate ...
sudo mkfs.ext4 -L pgdata /dev/mapper/vg-data
# or: /dev/vg/data symlink
lsblk -f /dev/mapper/vg-data
```

Format the LV (mapper node), not a raw PV member disk.

Capture UUID for fstab

```
UUID=$(sudo blkid -s UUID -o value /dev/sdb1)
echo "UUID=$UUID /mnt/data ext4 defaults,nofail 0 2"
# install into /etc/fstab after review, then:
sudo mount -a
```

nofail helps boot continue if an optional data disk is missing (tune to policy).

Whole-disk vs partition

```
# Prefer a partition table even for single-use data disks (clearer, ESP-friendly patterns).
# mkfs on whole /dev/sdb works but confuses some tools and dual-boot assumptions.
sudo fdisk /dev/sdb # create one partition first
sudo mkfs.ext4 -L data /dev/sdb1
```

Understanding Output

Successful `mkfs` prints progress and superblock/UUID details (ext*). Afterward:

```
lsblk -f /dev/sdb1
sudo blkid /dev/sdb1
```

Expect new FSTYPE, UUID, and optional LABEL. Old UUIDs in `fstab` become stale — update them.

Notes & Pitfalls

- **Irreversible** for practical purposes; snapshots/backups first.
- XFS: use `xfs_repair` offline; `fsck.xfs` does little.
- btrfs/zfs have their own create tools and operational models.
- Cloud disks: detach/attach order can rename `/dev/sdX` — use UUID in `fstab`.
- Encryption: `cryptsetup luksFormat` then `mkfs` on `/dev/mapper/...`, not the raw LUKS device path alone without opening.
- Filesystem choice: ext4 default on Ubuntu root; XFS common for large data; vfat for portable USB.

Related Commands

- `fdisk` / `parted` — partitions before `mkfs`
- `blkid` / `lsblk -f` — verify identity after format
- `tune2fs` — tune ext* without reformat
- `mkswap` / `swapon` — swap
- `mount` / `umount` / `fstab` — use the new FS
- `wipefs` — inspect/clear old signatures

Additional Resources

- `man mkfs`
- `man mkfs.ext4`
- `man mkfs.xfs`

fsck

Overview

`fsck` checks and optionally repairs filesystems. It is a front-end that dispatches to type-specific helpers (`fsck.ext4` → `e2fsck`, etc.). **Never run a repairing check on a mounted read-write filesystem** — unmount first, or use recovery/single-user media for the root FS. XFS is special: use `xfs_repair` offline; `fsck.xfs` is effectively a no-op.

Syntax

```
fsck [options] [-t type] [device|mountpoint...]
fsck.ext4 [options] device
e2fsck [options] device
```

Common Options

Option	Description
<code>-t type</code>	Filesystem type (or from <code>blkid</code> / <code>fstab</code>)
<code>-A</code>	All filesystems from <code>fstab</code>
<code>-R</code>	Skip root (with <code>-A</code>)
<code>-M</code>	Skip mounted filesystems
<code>-N</code>	Dry-run — show what would run
<code>-y</code>	Assume “yes” to all repairs (careful)
<code>-n</code>	Assume “no” / report-only (ext* via <code>e2fsck</code>)
<code>-f</code>	Force check even if clean
<code>-C</code>	Progress bar (ext*)
<code>-V</code>	Verbose
<code>-T</code>	Hide title

Safety

1. Identify the device: `lsblk -f`, `sudo blkid`.
2. Unmount: `sudo umount /dev/sdXN` (or boot rescue for root).
3. Prefer **report-only** (`-n` / `e2fsck -n`) before auto-yes (`-y`).

4. **XFS:** `sudo xfs_repair -n /dev/...` then `xfs_repair` without `-n`; do not expect `fsck.xfs` to repair.
5. Repairs can delete corrupt inodes/data — backups matter.
6. Do not `fsck` the live RW root; use recovery, live USB, or vendor maintenance mode.

Key Use Cases

1. After unclean shutdown / journal errors
2. Before reusing a suspect data disk
3. Boot-time automatic checks (fstab pass field / systemd)
4. Forced ext* check after resize or corruption reports

Examples with Explanations

Report-only on unmounted ext4

```
lsblk -f /dev/sdb1
sudo umount /mnt/data 2>/dev/null || true
sudo findmnt /dev/sdb1 && { echo "still mounted - abort"; exit 1; }
sudo fsck -n /dev/sdb1
# or explicit:
sudo e2fsck -n /dev/sdb1
```

`-n` opens read-only and reports problems without writing fixes.

Interactive repair

```
sudo umount /mnt/data
sudo fsck -f /dev/sdb1
```

Answer prompts carefully. `-f` forces a full check even if the FS is marked clean.

Auto-yes repair (after backup / when you accept loss)

```
sudo e2fsck -f -y /dev/sdb1
```

Convenient for automation and clearly disposable data; risky on irreplaceable volumes without backups.

Dry-run which helpers would run

```
sudo fsck -N -A
sudo fsck -N /dev/sdb1
```

Shows the planned `fsck.<type>` invocations without executing checks.

Force ext4 check with progress

```
sudo e2fsck -f -C 0 /dev/sdb1
```

Progress on file descriptor 0's tty-style reporting; useful on large volumes.

Root filesystem

```
# Boot Ubuntu recovery / live USB, find root device:
lsblk -f
# ensure root is not mounted RW, then:
sudo e2fsck -f /dev/nvme0n1p2
# systemd may run fsck on boot when dirty flags are set - check:
journalctl -b | grep -i fsck
```

Plan downtime. For cloud VMs, attach the disk to a rescue instance and fsck there.

XFS path

```
sudo umount /mnt/data
sudo xfs_repair -n /dev/sdb1    # no-modify scan
sudo xfs_repair /dev/sdb1      # repair
# NOT: fsck.xfs (does not repair)
```

Mount and verify after clean check

```
sudo fsck -n /dev/sdb1
sudo mount /dev/sdb1 /mnt/data
dmesg | tail
df -h /mnt/data
```

Understanding Output / Exit codes

Messages vary by helper. `e2fsck` prints phases (superblock, bitmaps, inodes, directory structure, ...).

Exit code is a **bitmask** (bits can combine):

Bit	Meaning
0	No errors
1	File system errors corrected
2	System should be rebooted
4	File system errors left uncorrected
8	Operational error
16	Usage or syntax error
32	Fsck canceled by user
128	Shared library error

```
sudo fsck -n /dev/sdb1
echo $?    # interpret with the table above
```

Notes & Pitfalls

- Mounted RW checks are unsafe; some tools refuse, others can destroy data.
- `fstab` sixth field controls boot fsck order/pass; 0 skips.
- `btrfs/zfs` use native check/scrub tools, not classic fsck workflows.
- Hardware faults: pair with `smartctl` — repairing FS won't fix a dying disk.
- Journal-only recovery often happens automatically on mount for `ext*`; full fsck is for deeper issues.
- `-y` can remove damaged files to restore consistency — that is success for fsck, data loss for you.

Related Commands

- `umount / mount` — offline requirement
- `tune2fs -l` — `ext*` superblock / mount counts
- `xfs_repair / btrfs check` — type-specific
- `smartctl` — disk health (`smartmontools`)
- `journalctl -b` — boot-time fsck messages
- `blkid / lsblk -f` — identify devices

Additional Resources

- `man fsck`
- `man e2fsck`
- `man xfs_repair`

mount

Overview

`mount` attaches a filesystem (block device, network share, bind path, tmpfs, ...) onto a directory in the single Linux directory tree. Persistent mounts belong in `/etc/fstab` or systemd `.mount` units; interactive `mount` is for live operations and troubleshooting.

Syntax

```
mount [-t type] [-o options] device dir
mount                                     # list mounts
mount -a                                 # all fstab entries due at boot
mount -o remount,opts dir
```

Common Options

Option	Description
<code>-t type</code>	Filesystem type (<code>ext4</code> , <code>xfs</code> , <code>nfs4</code> , <code>cifs</code> , <code>tmpfs</code> , <code>overlay</code> , ...)
<code>-o opts</code>	Mount options (<code>ro</code> , <code>noexec</code> , <code>nosuid</code> , <code>nodev</code> , <code>defaults</code> , ...)
<code>-a</code>	Mount all appropriate fstab entries
<code>-r / -w</code>	Read-only / read-write
<code>--bind / --rbind</code>	Bind mount (recursive)
<code>-v</code>	Verbose
<code>--source / --target</code>	Explicit long options

Key Use Cases

1. Attach disks and USB volumes
2. Mount NFS/CIFS shares
3. Bind mounts for chroots, containers, and rebuilds

4. Temporary tmpfs workspaces
5. Remount root or data with new options

Safety

- Wrong device → wrong data exposure. Verify with `lsblk -f` and `blkid` **before** mounting.
- Prefer `UUID=` / `LABEL=` in `fstab` over volatile `/dev/sdX` names.
- Network mounts can hang the shell on outage — consider `soft/timeo` options and separate automount units.
- Unmount cleanly (`umount`) before yanking USB or unplugging SAN LUNs.

Examples with Explanations

Inventory

```
findmnt
findmnt -t ext4,xfs
mount | column -t
findmnt -J | jq .           # if jq installed
```

Prefer `findmnt` for tree and JSON views.

Mount a partition by UUID

```
lsblk -f
sudo mkdir -p /mnt/data
sudo mount UUID=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx /mnt/data
df -h /mnt/data
```

Read-only mount (forensics / recovery)

```
sudo mount -o ro /dev/sdb1 /mnt/data
```

NFS

```
sudo mount -t nfs4 -o rw,hard,timeo=600 server.example:/export/path /mnt/nfs
```

Options trade durability vs hang behavior — know your storage SLA.

CIFS / SMB

```
sudo mount -t cifs //server/share /mnt/share \  
-o credentials=/root/.smbcred,uid=1000,gid=1000,ioccharset=utf8
```

Keep credentials files mode 600.

Bind mounts

```
sudo mount --bind /var/www /mnt/www-view  
sudo mount --rbind /home /mnt/chroot/home
```

Bind mounts share the same filesystem; **rbind** includes submounts.

tmpfs

```
sudo mount -t tmpfs -o size=1G,mode=0755 tmpfs /mnt/scratch
```

RAM-backed; contents vanish on unmount/reboot.

Remount

```
sudo mount -o remount,ro /  
sudo mount -o remount,rw /
```

Common in recovery environments.

fstab-driven

```
# /etc/fstab example:  
# UUID=... /mnt/data ext4 defaults,nofail 0 2  
sudo mount -a  
sudo mount /mnt/data  
findmnt /mnt/data
```

nofail avoids boot hangs if the disk is absent (laptops/USB).

Overlay (container-style)

```
sudo mount -t overlay overlay -o lowerdir=/base,upperdir=/upper,workdir=/work /merged
```

Used heavily by container engines; easy to get wrong — read `overlay` docs before production use.

Understanding Output

With no arguments, `mount` lists source, target, type, and options. Exit status non-zero means the mount failed (busy target, bad superblock, missing helper binary for NFS/CIFS, permission denied).

Notes & Pitfalls

- Helpers live as `mount.nfs`, `mount.cifs` — missing packages cause cryptic errors.
- “Target is busy” on `umount` → `lsof +f -- /mnt/data, fuser -vm /mnt/data`.
- Systemd may own the mount: `systemctl status mnt-data.mount`.
- `user/users` `fstab` options allow non-root mount/umount with restrictions.
- Mount namespaces (containers) mean `findmnt` inside a pod is not the host’s view.

Related Commands

- `umount` — detach
- `lsblk` / `blkid` — identify devices
- `findmnt` — rich listing
- `df` / `du` — space usage
- `losetup` — loop devices for images
- `fstab(5)` — persistence

Additional Resources

- `man mount`
- `man fstab`
- `man systemd.mount`

umount

Overview

`umount` detaches a mounted filesystem from the directory tree. Use it before safe device removal, before offline `fsck`/`mkfs`, or when tearing down bind/NFS mounts. Prefer the **mountpoint** path over the device node. If the target is busy, find holders with `lsof`/`fuser`/`findmnt` rather than forcing blindly.

Syntax

```
umount [options] <mountpoint|device>
umount -a [options]
```

Common Options

Option	Description
<code>-l</code>	Lazy unmount — detach now, cleanup when busy refs drop
<code>-f</code>	Force (mainly stale NFS); dangerous on local disks
<code>-r</code>	If unmount fails, try remount read-only
<code>-v</code>	Verbose
<code>-a</code>	Unmount all (respects filters; use carefully)
<code>-t type</code>	Limit by filesystem type (with <code>-a</code>)
<code>-R</code>	Recursive unmount of target and children (newer util-linux)

Key Use Cases

1. Clean detach of USB/data disks before unplug
2. Unmount before `fsck`, repartition, or `mkfs`
3. Clear busy or stale NFS mounts
4. Tear down bind mounts in chroots/containers/scripts

Safety

- Do **not** unmount / on a live system.
- Prefer stopping writers (services, shells with cwd on the mount) over **-f** on local media.
- Lazy (**-l**) hides problems; processes may still hold files until exit.
- Unmounting active databases or VMs can corrupt data — shut them down first.

Examples with Explanations

Unmount by mountpoint (preferred)

```
sudo umount /mnt/data
```

Matches how you mounted it; unambiguous when one device has multiple mounts.

Unmount by device

```
sudo umount /dev/sdb1
```

Works when you know the block device; fails if not mounted or still busy.

Confirm before and after

```
findmnt /mnt/data
lsblk -f /dev/sdb
sudo umount /mnt/data
findmnt /mnt/data || echo "unmounted"
```

Busy mount diagnosis

```
findmnt /mnt/data
sudo lsof +f -- /mnt/data
sudo fuser -vm /mnt/data
# common causes: shell cwd on mount, open logs, NFS locks, container bind
cd / # if your shell is the culprit
sudo umount /mnt/data
```

“Target is busy” almost always means open files or a process cwd under the tree.

Lazy unmount

```
sudo umount -l /mnt/data
```

Detaches from the namespace immediately; actual teardown when references drop. Use for stuck mounts when you understand leftover processes may still write until they exit.

Force NFS (last resort)

```
sudo umount -f /mnt/nfs  
# or lazy: sudo umount -l /mnt/nfs
```

For **stale** NFS when the server is gone. Avoid **-f** as a habit on local ext4/xfs.

Remount read-only fallback

```
sudo umount -r /mnt/data
```

If full unmount fails, try to freeze writes via remount-ro (util-linux behavior with **-r**).

Multiple mounts / bind trees

```
findmnt -R /mnt/data  
sudo umount -R /mnt/data # if supported: recursive  
# else unmount children first, then parent
```

Binds and nested mounts must be torn down from the leaves upward if recursive unmount is unavailable.

Before fsck

```
sudo umount /mnt/data  
sudo fsck -n /dev/sdb1
```

Never run repairing fsck on a read-write mounted filesystem.

Understanding Output

Success is silent (exit 0). Failures print to stderr:

Message	Typical cause
<code>target is busy</code>	Open files / cwd / mounts beneath
<code>not mounted</code>	Wrong path or already unmounted
<code>must be superuser</code>	Need root for most system mounts

Notes & Pitfalls

- Systemd may manage mounts as `*.mount` units — `systemctl stop mnt-data.mount` can be cleaner than raw `umount`.
- USB: unmount **then** wait for LED idle before unplug; otherwise risk corruption.
- `udisksctl unmount -b /dev/sdb1` is the desktop-friendly path on Ubuntu workstations.
- Open-but-deleted files on other filesystems don't block this unmount; busy means *this* mount's tree.
- In containers, mounts may be shared/slave — unmount semantics follow mount propagation.

Related Commands

- `mount` — attach filesystems
- `findmnt` — show what is mounted
- `lsblk` — device mountpoint map
- `lsof` / `fuser` — who is busy
- `fsck` — check offline filesystems
- `systemctl` — systemd mount units

Additional Resources

- `man umount`
- `man findmnt`

dd

Overview

dd copies and converts data at the block level. Operators use it to image disks/USBs, write ISO files to removable media, create sparse files, and dump raw device ranges. It is powerful and **easy to destroy the wrong disk** — always verify source and destination with `lsblk` first. For day-to-day file copies prefer `cp` or `rsync`; for cloning partitions with progress, many prefer `pv` + `dd` or dedicated tools (Clonezilla, `partclone`).

Syntax

```
dd if=INPUT of=OUTPUT [bs=BYTES] [count=N] [status=progress] [conv=OPTS] [oflag=OPTS] ...
```

Operands are `key=value` (GNU style). Order of operands generally does not matter.

Common Options

Option	Description
<code>if=FILE</code>	Input file/device (default: stdin)
<code>of=FILE</code>	Output file/device (default: stdout)
<code>bs=BYTES</code>	Read and write BYTES at a time (e.g. 4M)
<code>ibs= / obs=</code>	Separate input/output block sizes
<code>count=N</code>	Copy only N input blocks
<code>skip=N</code>	Skip N input blocks
<code>seek=N</code>	Skip N output blocks
<code>status=progress</code>	Periodic transfer stats (GNU)
<code>conv=fsync</code>	fsync output file before finish
<code>conv=noerror, sync</code>	Continue on read errors; pad bad blocks (imaging damaged media)
<code>conv=sparse</code>	Try to seek over zero blocks on output
<code>iflag=fullblock</code>	Accumulate full input blocks (safer pipes)
<code>oflag=direct</code>	Attempt O_DIRECT writes (device-dependent)
<code>oflag=sync</code>	Synchronous writes each block (slow, durable)

Size suffixes (GNU): K, M, G (1024-based); KB, MB, GB (1000-based). Check `man dd` for your coreutils version.

Safety

dd will overwrite the destination without asking. Wrong `of=` is a classic way to wipe a system disk.

1. **Identify devices first** — never guess `/dev/sdX` vs `/dev/nvme...`:

```
lsblk -o NAME,SIZE,TYPE,TRAN,MODEL,MOUNTPOINTS,SERIAL
# unplug/replug USB and re-run to see which node appears
```

2. **Confirm the USB/target is unmounted** before writing a whole disk image.
3. Prefer **status=progress** so a hung or wrong device is obvious sooner.
4. After writing boot media, run **sync** (or `conv=fsync`) before unplugging.
5. Prefer **read-only checks** (`if=` only, or write to a file first) when learning.
6. On multi-disk hosts, label targets and double-check `SERIAL/MODEL`.
7. **Do not** copy from/to a mounted filesystem's block device for “live backup” unless you know the consistency implications.

If unsure of the device node: **stop**. Physical write-protect switches and working on a spare machine reduce risk.

Examples with Explanations

List devices before any write

```
lsblk -o NAME,SIZE,TYPE,TRAN,MODEL,MOUNTPOINTS
# Example: USB stick shows as sdb, not sda (system disk)
```

Always map physical media → `/dev/...` before setting `of=`.

Write an ISO to a USB stick (destructive to the stick)

```
# 1) Identify stick - here assumed /dev/sdX (REPLACE X)
lsblk -o NAME,SIZE,TRAN,MODEL,MOUNTPOINTS

# 2) Unmount any auto-mounted partitions
sudo umount /dev/sdX* 2>/dev/null || true

# 3) Write whole-disk image
sudo dd if=ubuntu-24.04-desktop-amd64.iso of=/dev/sdX bs=4M status=progress conv=fsync oflag=

# 4) Ensure buffers flushed before yank
```

```
sync
```

`of=` is the **whole disk** (`/dev/sdX` / `/dev/nvme1n1`), not a partition, for hybrid ISOs. `bs=4M` is a practical throughput compromise; `conv=fsync` flushes on completion.

Create a raw disk image (backup a USB)

```
sudo umount /dev/sdX* 2>/dev/null || true
sudo dd if=/dev/sdX of=usb-backup-$(date +%F).img bs=4M status=progress conv=fsync
ls -lh usb-backup-*.img
```

Stores every byte of the device (including free space). Compress afterward if needed: `gzip -1 usb-backup.img`.

Restore an image to a USB

```
sudo umount /dev/sdX* 2>/dev/null || true
sudo dd if=usb-backup.img of=/dev/sdX bs=4M status=progress conv=fsync
sync
```

Destination size must be image size. Again: verify `of=` with `lsblk`.

Sparse empty file (test disk image)

```
dd if=/dev/zero of=disk.img bs=1M count=0 seek=1024
# or: truncate -s 1G disk.img
ls -lh disk.img
du -h disk.img          # sparse: little real space used
```

Useful for loop devices (`losetup`) and VM-style experiments without writing 1 GiB of zeros immediately.

Sample first megabyte (forensics / header peek)

```
sudo dd if=/dev/sdX bs=1M count=1 status=none | xxd | head
sudo dd if=/dev/sdX bs=512 count=1 of=mbr.bin
```

Read-only sampling of boot sectors or magic headers without imaging the whole device.

Damaged media (best-effort copy)

```
sudo dd if=/dev/sdX of=recovered.img bs=4M conv=noerror,sync status=progress
```

Continues past read errors and pads failed blocks. For serious recovery prefer `ddrescue` (`gddrescue` package) — better logging and retry logic.

Progress with pv (when installed)

```
sudo apt install pv      # if needed
sudo pv -tpreb ubuntu.iso | sudo dd of=/dev/sdX bs=4M conv=fsync
```

`pv` gives a clearer progress bar; still requires correct `of=`.

Wipe a USB free-space / reuse (destructive)

```
# Optional: overwrite with zeros (slow) or random
sudo dd if=/dev/zero of=/dev/sdX bs=4M status=progress conv=fsync
# Then re-partition / mkfs as needed
```

This destroys all partitions and data on the device. For secure erase of SSDs prefer vendor tools / `blkdiscard` where appropriate — sequential `dd` is not always the right model for flash.

Notes

- GNU `dd` on Ubuntu supports `status=progress`; some embedded `dd` builds do not.
- Block size affects speed more than correctness for full copies; very small `bs` is slow, huge `bs` can increase memory pressure.
- Writing an ISO with `cp/cat` to the device node also works on Linux; `dd` remains common in docs and scripts.
- USB path stability: prefer `/dev/disk/by-id/...` when scripting if nodes reorder across boots.
- Exit status is non-zero on incomplete/error copies — check it in scripts.
- `oflag=direct` can fail on some filesystems/devices; drop it if `dd` errors immediately.

Related Commands

- `lsblk` / `blkid` — identify devices safely
- `sync` — flush dirty pages after writes
- `umount` — unmount before imaging
- `losetup` — attach image files as loop devices
- `pv` — pipe progress

- `ddrescue` — resilient imaging of failing disks
- `wipefs` / `mkfs` — clear signatures / make filesystems after wipe
- `cp` / `rsync` — file-level copies (usually better for files)

Additional Resources

- `man dd`
- `info coreutils 'dd invocation'`

sync

Overview

`sync` flushes filesystem buffers: dirty page cache and pending metadata are written to storage. Use it before removing USB media, after bulk `dd/cp` to removable devices, or when you need a durable checkpoint before power loss. Modern kernels and `umount` already sync as part of a clean unmount; calling `sync` explicitly is still a useful operator habit for removable media and scripts.

Syntax

```
sync [options] [file...]
```

With no arguments, flushes all filesystems (global sync). With file arguments (GNU), flushes those files and the filesystems that hold them.

Common Options

Option	Description
<i>(none)</i>	Sync all (classic usage)
FILE...	Sync data for listed files (GNU coreutils)
<code>-d, --data</code>	Sync data only, not metadata (when supported)
<code>-f, --file-system</code>	Sync the filesystems that contain the files

BusyBox/`sync` on minimal systems may only support the no-argument form.

Safety

- `sync` is **not a substitute for unmounting**. Always `umount` (or `eject`) removable volumes when possible so the kernel finishes metadata updates cleanly.
- `sync` does **not** make a crash-consistent snapshot of multi-file databases by itself — use application flush/quiesce or LVM/ZFS snapshots for that class of problem.
- Calling `sync` under heavy write load can stall the system briefly while I/O drains — expected, not a hang in most cases.

Examples with Explanations

Flush everything (classic)

```
sync
```

Blocks until outstanding dirty data is queued to devices (and typically completed for the global case). Common after imaging USBs or large copies.

After writing a USB with dd

```
sudo dd if=image.iso of=/dev/sdX bs=4M status=progress conv=fsync
sync
# safely remove when activity LED is idle
```

conv=fsync already fsyncs the output; an extra sync is cheap insurance before unplug.

Sync specific file (GNU)

```
cp bigfile /media/usb/bigfile
sync /media/usb/bigfile
# or
sync -f /media/usb/bigfile
```

Narrows flush to the file/filesystem of interest instead of the whole machine.

Safe removable-media workflow

```
cp -a project/ /media/$USER/STICK/
sync
sudo umount /media/$USER/STICK
# or: udisksctl unmount -b /dev/sdX1
```

Copy → sync → unmount is the reliable order for thumb drives.

In scripts before reboot/power events

```
install -m 644 app.conf /etc/myapp/app.conf
sync
sudo systemctl restart myapp
```

Ensures config hit disk before a service that might race with a crash/reboot path.

Drop caches (admin debug — not for production routine)

```
sync  
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

`sync` first so `drop_caches` does not discard dirty data. Used for I/O benchmarks; not a general cleanup step.

Notes

- Journaling filesystems reduce risk of total corruption, but applications can still lose unflushed writes on power loss without `fsync/fdatasync`.
- `dd ... conv=fsync` and `cp` with certain flags interact with durability differently than a bare write + delayed writeback.
- Desktop “eject” / file-manager safe-remove typically unmounts (which syncs); CLI operators should mirror that with `umount`.
- Network filesystems: `sync` semantics depend on the protocol and mount options (`nfs`, `cifs`); local flush may not equal server durability.

Related Commands

- `dd` — block copy; pair with `sync/fsync`
- `umount` / `findmnt` — clean detach
- `mount -o sync` — mount with synchronous I/O (slow)
- `fsync` (syscall) / language APIs — per-fd durability
- `hdparm` / device caches — hardware write-cache caveats

Additional Resources

- `man sync`
- `man 2 fsync`

losetup

Overview

`losetup` associates regular files with **loop devices** (`/dev/loopN`) so the kernel can treat an image file like a block device—mount ISOs, inspect disk images, or access partitions inside a raw `.img`.

Syntax

```
losetup [options] [loopdev] [file]
losetup [options] -f file
```

Common Options

Option	Description
<code>-a, --all</code>	List active loop devices
<code>-f, --find</code>	Find first unused loop device
<code>-f --show</code>	Find, attach, print device path
<code>-d, --detach</code>	Detach
<code>-D, --detach-all</code>	Detach all (careful)
<code>-j file</code>	Show loops for file
<code>-r, --read-only</code>	Read-only
<code>-P, --partscan</code>	Scan partition table → <code>/dev/loopNp1...</code>
<code>-b SIZE</code>	Logical block size
<code>-o OFFSET</code>	Data offset
<code>--sizelimit</code>	Limit size
<code>-L, --nooverlap</code>	Refuse overlapping

Key Use Cases

1. Mount disk/ISO images
2. Access partitions inside a raw disk image
3. Filesystem experiments without real disks
4. Detach leftover loops after failed scripts

Examples with Explanations

Attach and list

```
sudo losetup -fP --show disk.img
sudo losetup -a
lsblk
```

Mount ISO

```
sudo losetup -f --show -r image.iso
# or directly:
sudo mount -o loop,ro image.iso /mnt/iso
```

Modern `mount -o loop` calls `losetup` for you.

Partitions inside image

```
sudo losetup -fP --show disk.img
lsblk /dev/loop0
sudo mount /dev/loop0p1 /mnt/p1
```

Detach

```
sudo umount /mnt/p1
sudo losetup -d /dev/loop0
sudo losetup -j disk.img
```

One-liner recipes

```
# Attach, print, and lsblk
dev=$(sudo losetup -fP --show disk.img); echo "$dev"; lsblk "$dev"

# Clean orphan loops (review first!)
sudo losetup -a
# sudo losetup -D # nuclear option
```

Notes & Pitfalls

- Always **umount before detach**.
- Sparse image files and sparse loop mappings can confuse free-space calculations.
- Permission: root required for setup.
- Snap/container environments may restrict loop devices.
- For qcow2 use `qemu-nbd` or `guestmount`, not raw `losetup` alone.

2026-relevant notes

- Cloud “custom image” workflows still use loop/nbd for injection of SSH keys and cloud-init.
- Prefer `udisksctl loop-setup` on desktops for user-friendly attach.
- `losetup -P + lsblk` is the standard recovery path for raw VM disks copied as files.

Related Commands

- `mount / umount` — filesystem attach
- `lsblk / blkid` — inspect
- `kpartx` — map partitions (alternative)
- `qemu-nbd` — qcow2 as block

Additional Resources

- `man losetup`
- `util-linux` documentation

duf

Overview

duf is a modern, colorful **disk usage** / **free space** viewer — a friendlier alternative to `df -h`. It groups local, network, special, and inaccessible filesystems with readable tables. Optional install (`apt install duf`, etc.). Keep `df` for scripts and minimal systems.

Syntax

```
duf [options] [device/path...]
```

Common Options

Option	Description
<code>-all</code>	Show all filesystems including pseudo
<code>-only local network fuse special loops binds</code>	Filter types
<code>-only-mp / -only-device</code>	Filter by mount/device strings
<code>-hide</code>	Hide types
<code>-output</code>	Choose columns
<code>-sort</code>	Sort key (<code>size</code> , <code>used</code> , <code>avail</code> , <code>usage</code> , ...)
<code>-json</code>	JSON output
<code>-style</code>	UI style
<code>-theme</code>	Color theme
<code>-width</code>	Width
<code>-inodes</code>	Inode view
<code>-avail-threshold / -usage-threshold</code>	Highlight thresholds

Confirm with `duf --help` for your version.

Examples with Explanations

Everyday

```
duf
duf -only local
duf /
duf -sort usage
```

Inodes

```
duf -inodes
df -ih
```

JSON

```
duf -json | jq .
```

Hide noise

```
duf -hide special,loops,binds
# or only local disks:
duf -only local
```

Compare with df

```
df -hT
duf
findmnt
```

Threshold awareness

```
duf -usage-threshold 0.8
```

Highlights nearly full mounts when supported.

Notes / Pitfalls

- Not portable to bare recovery images — know `df -hT`.
- Colors/unicode need a capable terminal.
- Bind mounts and snap loops can still clutter — filter them.
- JSON schema may change across versions.
- Network FS free space can be misleading (quotas, cloud buckets).

2026-relevant notes

- Excellent interactive upgrade alongside `dust/eza/bat`.
- Monitoring systems should still scrape `node_exporter/df` metrics, not `duf`.
- Btrfs/ZFS may need native tools for pool-level free space accuracy.

Related Commands

- `df` — portable free space
- `findmnt` — mount tree
- `du` / `dust` / `ncdu` — directory usage
- `lsblk` — block devices
- `btrfs filesystem usage` / `zpool list` — volume managers

Additional Resources

- `duf --help`
- [muesli/duf](#)

dust

Overview

dust is a modern **du alternative** that shows disk usage as a readable tree/bars, helping you find large directories quickly. Written in Rust; optional install. For portable scripts use **du -sh**; for interactive TUI browsing also consider **ncdu**.

Syntax

```
dust [options] [directory...]
```

Common Options

Option	Description
-d DEPTH, --depth	Max depth
-n N, --number-of-lines	Entries to show
-r, --reverse	Reverse natural order
-p, --full-paths	Full paths
-s, --apparent-size	Apparent size vs allocated
-i, --ignore-directory	Ignore path
-X file	Ignore from file
-e, --filter	Include only matching names
-v, --invert-filter	Invert filter
-c, --no-colors	Disable colors
-b, --no-percent-bars	No bars
-z, --min-size	Min size threshold
-t, --file-types	Group by filetype
-w, --terminal-width	Width

Check **dust --help** for current flags.

Examples with Explanations

Find big directories

```
dust
dust -d 2 /var
dust -n 30 /home
```

Apparent size

```
dust -s
du -sh --apparent-size *
```

Ignore noise

```
dust -i node_modules -i .git
dust -X .dustignore
```

File types

```
dust -t
```

Compare with du / ncd

```
du -h -d 1 /var | sort -h
dust -d 1 /var
ncdu /var
```

Min size filter

```
dust -z 100M /
```

Scripts: stick to du

```
du -sb /var/lib/docker
# dust is for humans
```

Notes / Pitfalls

- Permission errors on other users' dirs skew totals — run with appropriate rights.
- Hard links / snapshots / sparse files: understand apparent vs allocated.
- Network filesystems: slow; prefer scoped paths.
- Not preinstalled on servers.
- Colors in pipes may need `-c`.

2026-relevant notes

- Great laptop/dev host tool; production forensics still document `du/find`.
- Container disk growth: dust the overlay/graph roots carefully with host tools.
- Pair with `duf` for free space vs `dust` for “what ate it”.

Related Commands

- `du` — portable usage
- `ncdu` — interactive TUI
- `duf` / `df` — free space
- `find -size` — size predicates
- `btrfs filesystem du` — special FS accounting

Additional Resources

- `dust --help`
- [bootandy/dust](#)

parted

Overview

parted creates and modifies **partition tables** (GPT/MSDOS) on block devices. Prefer it for GPT disks and scripted partitioning; **fdisk** remains common for interactive MBR/GPT on small admin tasks. Mistakes destroy data — work on the correct device only.

Syntax

```
sudo parted [options] device [command...]  
sudo parted device print
```

Common Commands (interactive or CLI)

Command	Description
<code>print</code>	Show partition table
<code>mklabel gpt\ msdos</code>	Create new disklabel (wipes)
<code>mkpart</code>	Create partition
<code>rm N</code>	Delete partition number N
<code>resizepart</code>	Resize partition boundary
<code>unit GiB</code>	Display/input units
<code>align-check</code>	Alignment check

Safety

- **Double-check** `lsblk` / `by-id` paths before any `mklabel` or `rm`.
- `mklabel` destroys the existing table.
- After partitioning, create filesystems with `mkfs` and update `fstab` by UUID.
- On disks with LVM/RAID, coordinate with `pvcreate`/`mdadm` first.

Examples with Explanations

Print table

```
lsblk -o NAME,SIZE,TYPE,FSTYPE,UUID
sudo parted /dev/sdb print
sudo parted -l
```

GPT with one partition (scripted)

```
sudo parted -s /dev/sdb mklabel gpt
sudo parted -s /dev/sdb mkpart primary ext4 1MiB 100%
sudo parted /dev/sdb align-check optimal 1
sudo mkfs.ext4 /dev/sdb1
```

Interactive

```
sudo parted /dev/sdb
# (parted) unit GiB
# (parted) print
# (parted) quit
```

Notes & Pitfalls

- Kernel may need `partprobe` or a re-plug to re-read tables.
- NVMe partitions look like `/dev/nvme0n1p1` (note the `p`).
- Cloud disks often already have GPT — don't re-label casually.

Related Commands

- `fdisk` / `gdisk` — alternate partitioners
- `lsblk` / `blkid` — inventory
- `mkfs` / `mount` — after partitioning
- `wipefs` — clear signatures

Additional Resources

- `man parted`
- `info parted`

wipefs

Overview

wipefs probes and can **erase filesystem, RAID, or partition-table signatures** from a device without zeroing the entire disk. Useful before re-initializing disks that still claim to be members of LVM/MD/crypto volumes.

From util-linux.

Syntax

```
wipefs [options] device
sudo wipefs -a device
```

Common Options

Option	Description
-a	Erase all magic strings
-o offset	Erase specific offset
-n	Dry-run / no probe write (see man for version)
-p	From partition table probe
-t type	Limit to type
-f	Force

Safety

- This makes existing data **hard to mount**; it is destructive even though it is not a full shred.
- Confirm the device with `lsblk -f` and `blkid`.
- Prefer undoing LVM/MD membership with proper tools when the array is still healthy.

Examples with Explanations

Probe signatures

```
sudo wipefs /dev/sdb  
sudo wipefs -p /dev/sdb
```

Erase all signatures

```
sudo wipefs -a /dev/sdb  
sudo blkid /dev/sdb          # should be empty
```

After wipe — new filesystem

```
sudo parted -s /dev/sdb mklabel gpt  
sudo parted -s /dev/sdb mkpart primary 1MiB 100%  
sudo mkfs.xfs /dev/sdb1
```

Related Commands

- `blkid` — list signatures
- `lsblk -f` — overview
- `dd` — full overwrite (different goal)
- `cryptsetup` / `pvremove` / `mdadm` — structured teardown

Additional Resources

- `man wipefs`

smartctl

Overview

smartctl (from **smartmontools**) queries disk **S.M.A.R.T.** health data: overall assessment, attributes, error logs, and self-tests. Use it for predictive failure monitoring on HDDs/SSDs/NVMe (device-dependent).

```
sudo apt install smartmontools
```

Syntax

```
sudo smartctl [options] device
```

Common Options

Option	Description
<code>-a / -x</code>	All / extended SMART info
<code>-H</code>	Overall health PASS/FAIL
<code>-i</code>	Identity
<code>-A</code>	Vendor attributes table
<code>-l error</code>	Error log
<code>-l selftest</code>	Self-test log
<code>-t short\ long\ conveyance</code>	Start self-test
<code>-s on</code>	Enable SMART (if disabled)
<code>-d type</code>	Device type (raid, nvme, ...)

Safety

- Long self-tests stress disks — schedule during maintenance.
- On hardware RAID, you often need `-d megaraid,N` or controller tools; smartctl on `/dev/sdX` may talk to the virtual disk only.

- SMART “PASSED” is not a guarantee — watch reallocated sectors and media errors over time.

Examples with Explanations

Health summary

```
sudo smartctl -H /dev/sda
sudo smartctl -i /dev/sda
```

Full report

```
sudo smartctl -a /dev/sda | less
sudo smartctl -x /dev/nvme0 | less
```

Short self-test

```
sudo smartctl -t short /dev/sda
# wait, then:
sudo smartctl -l selftest /dev/sda
```

Watch failure predictors

```
sudo smartctl -A /dev/sda | egrep -i 'reall|pend|uncorrect|media|temperature'
```

Scan devices

```
sudo smartctl --scan
```

Related Commands

- `lsblk` / `nvme` — device inventory
- `dmesg` — I/O errors
- `hdparm` — identify / timings (older ATA focus)
- `smartd` — background monitoring daemon

Additional Resources

- `man smartctl`
- smartmontools FAQ

pvs

Overview

pvs reports **LVM physical volumes** (PVs) — disks or partitions initialized with **pvcreate** and used by volume groups. Part of the LVM reporting trio with **vgs** and **lvs**.

```
sudo apt install lvm2
```

Syntax

```
pvs [options] [PV...]
```

Common Options

Option	Description
<code>-o fields</code>	Custom columns
<code>-v</code>	Verbose
<code>--units h</code>	Human units
<code>-a</code>	All devices (incl. empty)
<code>--reportformat json</code>	JSON (newer)

Examples with Explanations

List physical volumes

```
sudo pvs
sudo pvs -o+pv_used,pv_free,dev_size
```

Initialize a new PV (related workflow)

```
sudo wipefs -a /dev/sdb1          # if reusing a disk - careful
sudo pvcreate /dev/sdb1
sudo pvs
```

Free space across PVs

```
sudo pvs --units g -o pv_name,vg_name,pv_size,pv_free
```

Notes & Pitfalls

- Whole-disk vs partition PVs: prefer a partition or carefully documented whole-disk use.
- Removing a PV requires migrating extents (`pvmove`) first if the VG has data there.

Related Commands

- `pvcreate` / `pvremove` / `pvresize`
- `vgs` / `lvs` — VG and LV reports
- `lsblk` — block layer view

Additional Resources

- `man pvs`
- `man lvm`

vgs

Overview

vgs reports **LVM volume groups** (VGs): how physical volumes are pooled and how much space is free for logical volumes. Use with **pvs** and **lvs** for a full LVM picture.

Syntax

```
vgs [options] [VG...]
```

Common Options

Option	Description
<code>-o fields</code>	Custom columns
<code>-v</code>	Verbose
<code>--units h</code>	Human-readable sizes
<code>-a</code>	Include hidden/internal where applicable

Examples with Explanations

List volume groups

```
sudo vgs
sudo vgs -o+vg_extent_size,vg_extent_count,vg_free_count
```

Create VG from PVs (workflow)

```
sudo vgcreate vg_data /dev/sdb1 /dev/sdc1
sudo vgs
```

Free space before creating an LV

```
sudo vgs --units g -o vg_name,vg_size,vg_free
```

Notes & Pitfalls

- Extent size is fixed at VG creation — plan large VGs carefully.
- Cluster/shared VGs need locking infrastructure (not covered here).

Related Commands

- `vgcreate` / `vgextend` / `vgreduce` / `vgremove`
- `pvs` / `lvs`
- `lvmdiskscan`

Additional Resources

- `man vgs`
- `man vgcreate`

lvs

Overview

`lvs` reports **LVM logical volumes** (LVs) — the block devices you usually format and mount (`/dev/VG/LV` or `/dev/mapper/VG-LV`). Check size, attributes (thin, cache, snapshot), and path before growing filesystems.

Syntax

```
lvs [options] [LV...]
```

Common Options

Option	Description
<code>-o fields</code>	Custom columns (<code>lv_path</code> , <code>lv_size</code> , <code>lv_attr</code> , ...)
<code>-a</code>	All LVs including internal
<code>--units h</code>	Human sizes
<code>-v</code>	Verbose

Examples with Explanations

List LVs

```
sudo lvs
sudo lvs -o lv_name,vg_name,lv_size,lv_path,lv_attr
```

Create and format an LV (workflow)

```
sudo lvcreate -n data -L 100G vg_data
sudo mkfs.ext4 /dev/vg_data/data
sudo mkdir -p /mnt/data
sudo mount /dev/vg_data/data /mnt/data
```

Grow an LV (then filesystem)

```
sudo lvextend -L +50G /dev/vg_data/data
sudo resize2fs /dev/vg_data/data      # ext*
# sudo xfs_growfs /mnt/data           # xfs, grow while mounted
```

Snapshots (classic thick)

```
sudo lvcreate -s -n data_snap -L 10G /dev/vg_data/data
sudo lvs
```

Understanding Output

`lv_attr` is a multi-character state field (open/mounted, snapshot, thin, ...). Paths under `/dev/mapper/` are commonly used in `fstab` via UUID of the **filesystem**, not the LV name alone.

Related Commands

- `lvcreate` / `lvextend` / `lvremove` / `lvrename`
- `pvs` / `vgs`
- `resize2fs` / `xfs_growfs`
- `findmnt`

Additional Resources

- `man lvs`
- `man lvcreate`

cryptsetup

Overview

`cryptsetup` manages **LUKS** (and other) disk encryption: format, open, close, and resize encrypted block devices. Standard tool for full-disk and data-partition encryption on Linux.

```
sudo apt install cryptsetup
```

Syntax

```
sudo cryptsetup [options] command device [args]
```

Common Commands

Command	Description
<code>luksFormat</code>	Initialize LUKS header (destroys data)
<code>open</code> / <code>luksOpen</code>	Map decrypted device under <code>/dev/mapper/</code>
<code>close</code> / <code>luksClose</code>	Remove mapping
<code>status</code>	Show mapper status
<code>luksAddKey</code> / <code>luksRemoveKey</code>	Manage passphrases/keys
<code>luksDump</code>	Show header metadata
<code>resize</code>	Resize LUKS mapping after LV growth

Safety

- `luksFormat` is irreversible without backups of the header and key material.
- Store recovery passphrases offline; header backups (`luksHeaderBackup`) matter.
- Suspend-to-disk with unlocked volumes has threat-model implications.
- Always confirm the target device with `lsblk` first.

Examples with Explanations

Format and open (data disk)

```
sudo cryptsetup luksFormat /dev/sdb1
sudo cryptsetup open /dev/sdb1 crypt_data
sudo mkfs.ext4 /dev/mapper/crypt_data
sudo mount /dev/mapper/crypt_data /mnt/data
```

Status and close

```
sudo cryptsetup status crypt_data
sudo umount /mnt/data
sudo cryptsetup close crypt_data
```

Add a second passphrase

```
sudo cryptsetup luksAddKey /dev/sdb1
sudo cryptsetup luksDump /dev/sdb1 | less
```

Header backup

```
sudo cryptsetup luksHeaderBackup /dev/sdb1 --header-backup-file ~/luks-sdb1.header
chmod 600 ~/luks-sdb1.header
```

fstab / crypttab sketch

```
# /etc/crypttab
# crypt_data UUID=... none luks,discard
# then fstab mounts /dev/mapper/crypt_data
```

Use UUIDs from `blkid` on the **LUKS** device for `crypttab`.

Notes & Pitfalls

- LUKS2 is the modern default on current distros.

- SSD discard/TRIM: `discard` option trades some security for wear-leveling — know your threat model.
- Nested LVM-on-LUKS or LUKS-on-LVM both exist; order affects unlock at boot.

Related Commands

- `lsblk` / `blkid` — find devices
- `dmsetup` — device-mapper layer
- `lvs` — often underneath or above LUKS
- `mount` / `umount`

Additional Resources

- `man cryptsetup`
- `crypttab(5)`

resize2fs

Overview

`resize2fs` grows or shrinks **ext2/ext3/ext4** filesystems. Growing online (mounted) is common after `lvextend` or partition expansion; shrinking requires an offline (unmounted) filesystem and extreme care.

Syntax

```
sudo resize2fs [options] device [size]
```

If `size` is omitted, the filesystem expands to the size of the underlying partition/LV.

Common Options

Option	Description
<code>-p</code>	Progress bars
<code>-f</code>	Force (rare)
<code>-M</code>	Shrink to minimum size (offline)
<code>-d flags</code>	Debug

Safety

- **Shrink** only when unmounted, after full backup and `e2fsck -f`.
- Grow the **block device first** (`lvextend`, `parted resizepart`), then `resize2fs`.
- XFS uses `xfs_growfs` (grow only) — not this tool.
- Btrfs/ZFS have their own grow tools.

Examples with Explanations

Grow to fill LV after lvextend

```
sudo lvextend -L +20G /dev/vg0/root
sudo resize2fs /dev/vg0/root
df -h /
```

Explicit size

```
sudo resize2fs /dev/vg0/data 50G
```

Offline shrink (dangerous — backup first)

```
sudo umount /mnt/data
sudo e2fsck -f /dev/vg0/data
sudo resize2fs /dev/vg0/data 30G
# then shrink LV if desired (lvreduce), carefully matching sizes
```

Notes & Pitfalls

- Kernel/online resize support is standard for grow on ext4; shrink is offline-only.
- Cloud “growpart” + resize2fs is a common first-boot pattern.
- Always match units with LVM tools (g vs G conventions).

Related Commands

- `lvextend` / `lvs`
- `e2fsck` / `tune2fs`
- `xfs_growfs` — XFS
- `findmnt` / `df`

Additional Resources

- `man resize2fs`

tune2fs

Overview

tune2fs adjusts tunable parameters on **ext2/ext3/ext4** filesystems: labels, UUID, mount-count checks, reserved blocks, feature flags, and more. Inspect with **tune2fs -l** before changing anything.

Syntax

```
sudo tune2fs [options] device
```

Common Options

Option	Description
-l	List current superblock contents
-L label	Set volume label
-U UUID\ random\ time	Set UUID
-c max-mount-counts	Max mounts between fsck (-1 disables)
-i interval	Max time between fsck
-m percent	Reserved blocks % for root
-r blocks	Reserved block count
-o mount-options	Default mount options
-0 features	Set/clear features (careful)

Safety

- Wrong device ruins the wrong filesystem — confirm with **blkid / lsblk -f**.
- Feature flag changes (-0) can make a volume unmountable on older kernels.
- Reserved block percentage affects usable free space for non-root users.

Examples with Explanations

Inspect

```
sudo tune2fs -l /dev/vg0/root | egrep -i 'label|uuid|block count|reserved|mount count|check'
```

Label and UUID

```
sudo tune2fs -L ROOT /dev/vg0/root
sudo tune2fs -U random /dev/sdb1
sudo blkid /dev/sdb1
# update fstab if UUID changed!
```

Reserved space (big data disks)

```
sudo tune2fs -m 1 /dev/vg0/data      # 1% reserved instead of default ~5%
```

Disable frequent forced fsck by mount count

```
sudo tune2fs -c -1 /dev/vg0/data
sudo tune2fs -i 0 /dev/vg0/data
```

Still run intentional checks after crashes.

Related Commands

- `blkid` / `e2label` — labels/UUIDs
- `resize2fs` — size changes
- `e2fsck` / `fsck` — checks
- `dumpe2fs` — alternate dump

Additional Resources

- `man tune2fs`

ncdu

Overview

ncdu (NCurses Disk Usage) is an interactive **du** for finding what consumes space. Faster feedback than reading raw **du** trees; great on full root filesystems.

```
sudo apt install ncdu
```

Syntax

```
ncdu [options] [path]
```

Common Options

Option	Description
-x	One filesystem only
-r	Read-only (no delete)
-o file	Export scan
-f file	Load export
-e	Extended info (more stat calls)
--exclude pattern	Skip paths

Examples with Explanations

Scan root (one FS)

```
sudo ncdu -x /
```

Navigate with arrows; **d** deletes (if not **-r**); **q** quits.

Home directory

```
ncdu -x "$HOME"
```

Export for later / remote analysis

```
sudo ncdu -x -o /tmp/root.ncdu /  
ncdu -f /tmp/root.ncdu
```

Notes & Pitfalls

- Needs read permission; use `sudo` for system trees.
- Exclude bind mounts and huge network mounts with `-x / --exclude`.
- Deleting from `ncdu` is still delete — careful on production.

Related Commands

- `du / dust / duf` — other usage views
- `df` — filesystem free space
- `find` — age-based cleanup

Additional Resources

- `man ncdu`

Terminals and Multiplexers

Intro

Intro

Keep shells alive across disconnects, split panes, and share sessions. `tmux` is the usual default; `screen` remains common on older hosts.

Commands in this part

Command	Role
<code>tmux</code>	<code>tmux</code> is a terminal multiplexer: multiple windows and panes inside detachable sessions that survive SSH disconnects.
<code>screen</code>	<code>screen</code> is the classic terminal multiplexer: detachable sessions that survive hangups.
<code>byobu</code>	The <code>byobu</code> command is a text-based window manager and terminal multiplexer.
<code>tmate</code>	<code>tmate</code> is a fork of <code>tmux</code> focused on instant terminal sharing.
<code>abduco</code>	<code>abduco</code> is a minimal session attach/detach tool: it runs a process under a controlling terminal session you can...

Suggested starting points

1. Daily driver: `tmux` (sessions, windows, panes).
2. Legacy hosts: `screen`.
3. Convenience wrappers: `byobu`.
4. Remote pair/share: `tmate`; minimal attach tools: `abduco`.

Related parts

- Networking — `ssh` into the host first
- Processes and jobs — reattach instead of losing long jobs

- Shell commands — history inside each pane

Continue with the individual command pages in this part.

tmux

Overview

tmux is a **terminal multiplexer**: multiple windows and panes inside detachable sessions that survive SSH disconnects. Prefer it over **nohup** for interactive long-lived remote work, and over ad-hoc background jobs when you need to reattach and see output. For production services still prefer **systemd**.

Syntax

```
tmux [command] [options]
tmux [-L socket-name] [-S socket-path] [-f file]
```

Most daily use is either bare **tmux** or a short subcommand (**new**, **attach**, **ls**, **kill-session**).

Install (Ubuntu)

```
sudo apt update
sudo apt install tmux
tmux -V
```

Prefix key

Default prefix: **Ctrl-b**, then a command key. (Many people rebind to Ctrl-a.)

After prefix	Action
c	New window
,	Rename window
n / p	Next / previous window
0–9	Select window by index
w	Window list
&	Kill window
%	Split pane left/right
"	Split pane top/bottom

After prefix	Action
o / arrow keys	Cycle / move between panes
;	Last pane
x	Kill pane (confirm)
z	Zoom toggle (pane fullscreen)
{ / }	Swap panes
Space	Cycle pane layouts
d	Detach session
s	Session list
\$	Rename session
[	Copy / scroll mode
]	Paste buffer
?	List keys
:	Command prompt

Common CLI commands

Command	Description
<code>tmux</code>	New session (default name)
<code>tmux new -s NAME</code>	Named session
<code>tmux new -d -s NAME [cmd]</code>	Detached session (optional command)
<code>tmux ls</code>	List sessions
<code>tmux attach -t NAME</code>	Attach
<code>tmux attach -d -t NAME</code>	Attach and detach others
<code>tmux switch -t NAME</code>	Switch (from inside tmux)
<code>tmux kill-session -t NAME</code>	Kill session
<code>tmux kill-server</code>	Kill all sessions (careful)
<code>tmux list-windows -t NAME</code>	Windows in session

Key Use Cases

1. Survive laptop sleep / SSH drops during deploys
2. Pair multiple panes (logs + editor + shell)
3. Long builds or trainers you can reattach to
4. Named sessions per project (`tmux new -s api`)

Examples with Explanations

Example: create, detach, reattach

```
tmux new -s deploy
# ... work ...
# Ctrl-b d
tmux ls
tmux attach -t deploy
# short form:
tmux a -t deploy
```

Core workflow. Detach leaves processes running on the server.

Example: start detached job session

```
tmux new -d -s train 'python3 train.py 2>&1 | tee train.log'
tmux attach -t train
```

Session exists immediately; attach when you want the TTY.

Example: nested SSH-safe habit

```
ssh host
tmux new -s work || tmux attach -t work
```

Create-or-attach pattern for a daily session.

Example: splits for ops

Inside a session:

1. Ctrl-b % — vertical split (side by side)
2. Ctrl-b " — horizontal split
3. Ctrl-b z — zoom the pane you care about
4. Run `journalctl -fu nginx` in one pane, shell in another

Example: scripted window layout

```
tmux new -d -s app
tmux rename-window -t app:0 shell
tmux new-window -t app -n logs 'journalctl -f'
tmux new-window -t app -n top 'htop'
```

```
tmux split-window -h -t app:shell
tmux attach -t app
```

Good for standardizing a “ops dashboard” session.

Example: send keys to a session (automation)

```
tmux new -d -s patch
tmux send-keys -t patch 'sudo apt update && sudo apt upgrade' C-m
```

Use sparingly; prefer scripts over pseudo-interactive automation when possible.

Example: shared socket / pair-ish access

```
tmux -S /tmp/pair.sock new -s pair
# other user (permissions permitting):
tmux -S /tmp/pair.sock attach -t pair
```

For real pairing, tools like `tmate` or proper access control may be better.

Example: minimal `~/.tmux.conf`

```
set -g mouse on
set -g history-limit 50000
set -g base-index 1
setw -g pane-base-index 1
set -g renumber-windows on
# rebind prefix to Ctrl-a (optional):
# unbind C-b
# set -g prefix C-a
# bind C-a send-prefix
```

Reload from inside tmux: `Ctrl-b : then source-file ~/.tmux.conf`.

Example: copy mode (scrollback)

`Ctrl-b` [→ move with arrows/PageUp → Space begin selection → Enter copy (bindings vary with mode-keys vi/emacs) → `Ctrl-b`] paste.

With mouse on, scroll and select often work directly.

Safety

- `tmux kill-server` ends **all** sessions for that server socket — confirm `tmux ls` first.
- Detached sessions still consume resources; clean up dead project sessions.
- Do not run secrets printing in shared sessions.
- Nested tmux (local + remote) confuses prefixes — use a different remote prefix or SSH directly into remote tmux only.

Notes & Pitfalls

- Environment variables are captured at **session start**; re-login SSH agent sockets often need `tmux setenv` or plugin tooling after laptop sleep.
- `TERM/COLORTERM` mismatches cause weird colors — prefer `tmux -2` legacy force or correct `terminal-overrides` in conf for modern terminals.
- Clipboard integration needs `xclip/wl-clipboard` and conf bindings on desktop Linux.
- Version differences: Ubuntu LTS tmux may lag plugin ecosystem features — check `tmux -V`.
- Prefer one session per task rather than infinite windows without names.

Related Commands

- `screen` — older multiplexer
- `byobu` — profile/enhancement layer over `tmux/screen`
- `tmate` — easy remote sharing
- `abduco` / `dtach` — minimal detach only
- `nohup` / `systemd-run` — non-interactive longevity
- `ssh` — usual transport

Additional Resources

- `man tmux`
- `tmux list-keys` / `tmux list-commands`

screen

Overview

screen is the classic terminal multiplexer: detachable sessions that survive hangups. It is still available on Ubuntu and many rescue environments, but **new muscle memory should prefer `tmux`** unless you are on a host that only has **screen** or you maintain legacy automation. Concepts map closely: sessions, windows, detach/attach.

Syntax

```
screen [options] [command [args]]
screen -S name
screen -r [name]
screen -ls
```

Install (Ubuntu)

```
sudo apt update
sudo apt install screen
screen -v
```

Prefix key

Default prefix: **Ctrl-a**, then a command key. (Note: conflicts with readline “beginning of line” — press **Ctrl-a a** to send literal Ctrl-a.)

After prefix	Action
c	New window
n / p	Next / previous window
"	Window list
A	Rename window
k	Kill window
S	Split horizontal (regions)
	Split vertical (modern screen; if unbound, use conf)

After prefix	Action
Tab	Focus next region
X	Remove region
d	Detach
[/ Esc	Copy / scroll mode
]	Paste
?	Help
:	Command line

Common CLI options

Option	Description
-S name	Session name
-dmS name cmd	Start detached named session running cmd
-ls / -list	List sessions
-r name	Reattach
-R	Reattach if exists else create
-d -r name	Detach elsewhere then reattach here
-x name	Multi-display attach (share session)
-X stuff '...'	Send input to a session
-wipe	Remove dead session entries

Key Use Cases

1. Keep a shell alive across SSH drops on legacy hosts
2. Detached long runners where tmux is not installed
3. Sharing a session with -x for quick co-debugging
4. Rescue environments that ship `screen` only

Examples with Explanations

Example: create, detach, resume

```
screen -S deploy
# ... work ...
# Ctrl-a d
screen -ls
screen -r deploy
```

If only one detached session exists, `screen -r` is enough.

Example: create-or-attach

```
screen -D -RR work
```

Aggressive “get me a session named work” pattern used in older docs; know it may detach others.

Example: start detached command

```
screen -dmS train bash -lc 'python3 train.py 2>&1 | tee train.log'
screen -r train
```

-dmS = detached + create + name. Wrap with `bash -lc` when you need login profile or pipelines.

Example: list and wipe zombies

```
screen -ls
screen -wipe
```

Dead sessions after crashes leave “Dead ???” entries; wipe cleans the registry.

Example: multi-attach

```
screen -x deploy
```

Second terminal joins the same session (both see the same windows).

Example: send commands from outside

```
screen -S deploy -X stuff 'systemctl status nginx^M'
```

^M is Enter. Fragile compared to real scripts — use for glue only.

Example: log session output

Inside screen: `Ctrl-a H` toggles logging to `screenlog.n` in the cwd (default). Or start with:

```
screen -L -S logged
```

Example: minimal ~/.screenrc

```
startup_message off
defscrollback 10000
shell -$SHELL
# harder status line:
hardstatus alwayslastline
hardstatus string '%{= kG}[ %{G}%H %{g}][%= %{= kw}??%-Lw%?%{r}(%{W}%n*%f%t%?(%u)%?%{r})%{w}
```

Example: vertical split note

Classic `screen` was weak at tiling compared to `tmux`. Recent GNU `screen` supports vertical splits (`Ctrl-a |` when bound). If your Ubuntu version feels awkward for panes, that is a normal reason to switch to `tmux`.

Safety

- Killing the wrong session loses unsaved editor buffers in that session.
- `screen -X` can inject input into interactive tools — dangerous on shared sessions.
- Multi-attach (`-x`) shares keystrokes; coordinate before typing production commands.
- Session sockets live under hard-to-guess paths but are still local privilege boundaries — do not relax directory permissions carelessly.

Notes & Pitfalls

- **Prefix conflicts** with Emacs/readline `Ctrl-a` — muscle memory cost is the #1 complaint.
- Escape detachment: `Ctrl-a d` vs nested `screen/ssh` needs extra care (`Ctrl-a a d` patterns).
- Environment variable staleness after reattach (`SSH_AUTH_SOCK`) same class of problem as `tmux`.
- Prefer `tmux` for new documentation and training unless constrained.
- BusyBox or tiny systems may ship neither; fall back to `nohup/systemd`.

Related Commands

- `tmux` — modern default multiplexer
- `byobu` — friendly wrapper (can use `screen` backend)
- `tmate` — remote pairing
- `abduco` / `dtach` — detach without windows
- `nohup` — minimal HUP immunity

Additional Resources

- `man screen`
- `Ctrl-a ?` inside a session

byobu

Overview

The `byobu` command is a text-based window manager and terminal multiplexer. It's a wrapper for `tmux`/`screen` that provides enhanced features and an easier interface.

Syntax

```
byobu [options] [command]
```

Common Options

Option	Description
-S	Start in screen mode
-T	Start in tmux mode
-v	Version info
-h	Show help
--help	Detailed help
-l	List sessions
-L	Select backend
-c command	Run command
-f	Frontend
-p profile	Use profile

Function Keys

Key	Action
F2	Create window
F3	Previous window
F4	Next window
F5	Reload profile
F6	Detach session
F7	Enter scrollbar

Key	Action
F8	Rename window
F9	Configuration menu
F12	Lock session
Shift-F2	Split vertical

Key Use Cases

1. Session management
2. System monitoring
3. Remote work
4. Project organization
5. Server administration

Examples with Explanations

Example 1: Start Session

```
byobu
```

Start new session

Example 2: Named Session

```
byobu new -s mysession
```

Create named session

Example 3: List Sessions

```
byobu list-sessions
```

Show running sessions

Common Usage Patterns

1. Basic start:

```
byobu
```

2. Attach session:

```
byobu attach
```

3. Kill session:

```
byobu kill-session
```

Security Considerations

1. Session security
2. Remote access
3. Multi-user mode
4. Screen locking
5. SSH integration

Related Commands

- `tmux` - Terminal multiplexer
- `screen` - Terminal multiplexer
- `tmate` - Sharing tool
- `ssh` - Secure shell
- `dtach` - Session detachment

Additional Resources

- [Byobu Documentation](#)
- [Usage Guide](#)
- [System Administration](#)

Best Practices

1. Use status info
2. Configure profiles
3. Custom keybindings
4. Regular updates

5. Session naming

Configuration

1. Status notifications
2. Window layout
3. Color schemes
4. Key bindings
5. Profiles

Troubleshooting

1. Backend issues
2. Display problems
3. Key conflicts
4. Profile errors
5. Performance

tmate

Overview

tmate is a fork of **tmux** focused on **instant terminal sharing**. It creates a session and prints SSH/web URLs so collaborators can attach remotely. Great for pair debugging; treat access tokens as secrets. For local-only multiplexing without sharing, use stock **tmux**.

Syntax

```
tmate [tmux-compatible options]
tmate show-messages
tmate wait tmate-ready
```

Most session control mirrors **tmux** keybindings and commands.

Common usage patterns

Action	Example
Start shared session	<code>tmate</code>
Wait until ready	<code>tmate wait tmate-ready</code>
Print connection info	<code>tmate show-messages</code> / display-panes messages
Detach	<code>Ctrl-b d</code> (default prefix like tmux)
Local config	<code>~/.tmate.conf</code>

Examples with Explanations

Start a session

```
tmate
# note the ssh and web URLs printed
tmate wait tmate-ready
tmate show-messages
```

Capture SSH URL for a colleague

```
tmate -F show-messages | grep 'ssh session:'  
# or scripted:  
tmate -S /tmp/tmate.sock new-session -d  
tmate -S /tmp/tmate.sock wait tmate-ready  
tmate -S /tmp/tmate.sock display -p '#{tmate_ssh}'
```

Read-only vs read-write links

tmate provides different access strings (RO/RW). Share **read-only** by default; only give RW to trusted peers.

Config sketch

```
# ~/.tmate.conf  
set -g tmate-server-host ssh.tmate.io  
# org-hosted servers possible for privacy
```

End session

```
# exit shells or kill-session  
tmate kill-server
```

Prefer local tmux when not sharing

```
tmux new -s solo
```

Notes / Pitfalls

- Default public relay means session metadata/paths may transit third parties — use a **self-hosted** tmate server for sensitive work.
- Anyone with the RW SSH string has full terminal control — treat like a root password.
- Corporate networks may block outbound SSH to tmate hosts.
- Still tmux-based: know prefix keys, panes, and copy-mode.
- Don't leave orphan shared sessions on bastion hosts.

2026-relevant notes

- Alternatives: VS Code Live Share, SSH with `tmux` + explicit user accounts, commercial pair tools.
- For regulated environments, mandate self-hosted relays or forbid `tmate`.
- Rotate by killing sessions after the pairing call ends.

Related Commands

- `tmux` — local multiplexer foundation
- `screen` — classic alternative
- `ssh` — direct remote access
- `script` / `asciinema` — record sessions without live share

Additional Resources

- `man tmate`, tmate.io documentation

abduco

Overview

abduco is a minimal **session attach/detach** tool: it runs a process under a controlling terminal session you can disconnect from and reattach to later. Lighter than full multiplexers (**tmux**, **screen**) — no window manager, just detachable sessions. Often paired with **dvtm** for tiling inside the session.

Optional package; not installed by default on most distros.

Syntax

```
abduco [options] [-e key] [-r] name
abduco -A name [command...]
abduco -l
```

Common Options

Option	Description
-A name	Attach or create session name
-a name	Attach to existing session
-c name command	Create session running command
-e key	Escape key (default Ctrl-\)
-r	Read-only attach
-l	List sessions
-n	Create but don't attach
-f	Force (see man for attach behavior)
-q	Quiet

Exact flags can vary slightly by version — check **abduco -h**.

Examples with Explanations

Create / attach

```
abduco -A work
# runs $SHELL by default in a new session if needed
```

Run a specific command

```
abduco -c build make -j$(nproc)
abduco -a build
```

List and reattach

```
abduco -l
abduco -a work
```

Detach

```
Ctrl-\    (default detach key - confirm with man/help)
```

Or close the SSH client carefully knowing the session survives.

Read-only observe

```
abduco -r -a work
```

Pair with dvtm

```
abduco -A dev dvtm
```

dvtm provides tiling; abduco provides detach.

vs tmux

```
# tmux: windows, panes, scripting, ecosystem
tmux new -s work
# abduco: tiny attach/detach
abduco -A work
```

Notes / Pitfalls

- Must be installed; rare on enterprise minimal images.
- Escape key conflicts — rebind with `-e` if needed.
- Not a full replacement for tmux scripting/automation.
- Session socket location depends on build (`ABDUCCO_SOCKET_DIR` / runtime dir).
- Sharing sessions multi-user needs careful permissions (prefer tmux/socket policies or separate tools).

2026-relevant notes

- Nice for constrained environments where tmux feels heavy, or for teaching detach basics.
- For production admin work, **tmux** or **systemd** still dominate.
- Combine with mosh/SSH keepalive for flaky links.

Related Commands

- `tmux` / `screen` — full multiplexers
- `dvtm` — tiling wm for terminal
- `dtach` — similar detach tool
- `nohup` / `disown` — weaker persistence
- `systemd-run` — transient services

Additional Resources

- `abduco -h`, project docs (martanne/abduco)

Text and Pipes

Intro

Intro

Filter, reshape, and measure text streams — the core of shell pipelines. Line tools (**grep/sed/awk**) plus structured JSON (**jq**) and progress (**pv**).

Commands in this part

Command	Role
grep sed	grep searches input lines for a pattern. sed (stream editor) transforms text line by line — substitute, delete, print, and simple inserts — without opening...
awk	awk is a pattern-action language for column-oriented text.
cut	The cut command extracts specific columns or fields from lines of text.
sort	sort orders lines of text (or selected fields) and can merge pre-sorted files.
uniq	uniq reports or filters adjacent duplicate lines.
tr	The tr (translate) command translates or deletes characters from standard input.
head	The head command displays the first lines of files or input streams.
tail	The tail command displays the last lines of files or input streams.
wc	The wc (word count) command counts lines, words, characters, and bytes in files or input streams.
tee	tee reads stdin and writes it to both standard output and one or more files.
xargs	xargs builds and executes command lines from standard input.
diff	The diff command compares files line by line and displays the differences.
comm	comm compares two sorted files line by line and outputs three columns: lines only in file1, only in file2, and...

Command	Role
<code>paste</code>	paste merges lines of files side by side, writing corresponding lines separated by tabs (or another delimiter).
<code>column</code>	column formats text into aligned columns or a table.
<code>jq</code>	jq is a command-line JSON processor.
<code>pv</code>	pv (Pipe Viewer) shows a progress bar, throughput, and ETA for data moving through a pipeline.

Suggested starting points

1. Filter: `grep` (or `rg` in Files for recursive code search).
2. Rewrite: `sed`; fields/reports: `awk`.
3. Columns and sets: `cut`, `sort`, `uniq`, `comm`, `paste`, `column`.
4. Edges of streams: `head`, `tail`, `tee`, `xargs`.
5. Compare: `diff`; JSON: `jq`; progress: `pv`.

Related parts

- Files and paths — sources of text
- Shell commands — quoting and `printf`
- Scripting → Bash (this book) — programming beyond one-liners

Continue with the individual command pages in this part.

grep

Overview

grep searches input lines for a pattern. GNU grep supports basic/extended/Perl-like regexes, recursive directory search, and rich context controls. Still the default everywhere; rg (ripgrep) is a popular faster alternative when installed.

Syntax

```
grep [OPTIONS] PATTERN [FILE...]  
grep [OPTIONS] -e PATTERN ... [FILE...]  
command | grep [OPTIONS] PATTERN
```

Common Options

Option	Description
-E	Extended regex (ERE)
-F	Fixed strings (no regex)
-P	Perl-compatible regex (PCRE) if built-in
-i	Case-insensitive
-v	Invert match
-n	Line numbers
-H / -h	With / without filename
-r / -R	Recursive (follow / include symlinks differently)
-I	Skip binary
-l / -L	Files with / without matches
-c	Count matches per file
-A/-B/-C n	Context after/before/both
-w	Whole word
-o	Only the match
-m n	Stop after n matches per file
--color=auto	Highlight
-q	Silent (exit status only)

Key Use Cases

1. Filter logs and command output
2. Code/config search in trees
3. Scripting success/failure via exit status
4. Extract substrings with `-o`

Examples with Explanations

Basic

```
grep error /var/log/syslog  
grep -i error /var/log/syslog
```

Extended regex

```
grep -E 'error|warn|fail' app.log
```

Fixed string (fast, safe)

```
grep -F 'https://example.com/path?x=1' urls.txt
```

Recursive code search

```
grep -RIn --exclude-dir=.git --exclude-dir=node_modules 'TODO' .
```

Context

```
grep -n -C3 'OutOfMemory' heap.log
```

Files that match

```
grep -rl 'Listen 80' /etc/apache2 2>/dev/null
```


Exit status in scripts

```
if grep -q 'ready' /tmp/status; then
    echo ok
fi
```

Only matching part

```
grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' access.log | sort -u
```

Invert

```
grep -v '^#' /etc/ssh/sshd_config | grep -v '^$'
```

Understanding Output

Default: matching lines. With multiple files, lines are prefixed by filename. Exit codes: 0 match, 1 no match, 2 error.

Common Usage Patterns

Pipe from journalctl

```
journalctl -u nginx --since today | grep -i error
```

Count HTTP status codes

```
grep -oE 'HTTP/[0-9.]+ ' [0-9]{3}' access.log | awk '{print $NF}' | sort | uniq -c
```

Notes & Pitfalls

- Quote patterns to stop shell globbing.
- Basic regex treats +/| differently than -E.
- Binary matches print “Binary file ... matches”; use -a or -I.

- For huge repos, consider `rg` or `git grep`.

Related Commands

- `rg` — ripgrep (if installed)
- `ack` / `ag` — other code search tools
- `sed` / `awk` — transform, not just filter
- `find ... -exec grep` — older recursive pattern
- `journalctl` / `dmesg` — log sources

Additional Resources

- `man grep`
- GNU grep manual

sed

Overview

sed (stream editor) transforms text **line by line** — substitute, delete, print, and simple inserts — without opening an interactive editor. GNU **sed** is standard on Ubuntu/Debian. Prefer **sed** for mechanical rewrites; use **awk** for field logic and an editor for multi-line structural edits.

Syntax

```
sed [options] 'script' [file...]  
sed [options] -e 'script' -e 'script' [file...]  
sed [options] -f script.sed [file...]  
... | sed 'script'
```

Common Options

Option	Description
-n	Quiet; only explicit p prints
-E / -r	Extended regex (ERE)
-i[SUFFIX]	In-place edit (optional backup suffix)
-e	Add a script expression
-f file	Read script from file
-z	NUL-separated “lines” (GNU; useful with find -print0)

Common script commands

Cmd	Meaning
s/pat/repl/flags	Substitute (g global, i ignore case, p print, 2 Nth only)
d	Delete line
p	Print line
a\ / i\ / c\	Append / insert / change (multi-line form in scripts)
q / Q	Quit (GNU Q without printing pattern space)
y/src/dst/	Transliterate like tr

Cmd	Meaning
=	Print line number

Safety

- `sed -i` rewrites files — always keep backups (`-i.bak`) until the expression is trusted.
- Prefer **single quotes** around scripts so the shell does not expand `$` and ```.
- Never run `sed -i` on binary files or unknown encodings.
- On macOS, `-i` requires a mandatory suffix argument — scripts written for GNU `sed` may break there.

Key Use Cases

1. Bulk string replace in config/logs
2. Strip comments/blank lines
3. Extract a capture group (print-only substitute)
4. Normalize line endings / trailing whitespace
5. Quick in-place flips for feature flags

Examples with Explanations

Substitute first match per line

```
sed 's/foo/bar/' file.txt
```

Only the first `foo` on each line becomes `bar`.

Global substitute

```
sed 's/foo/bar/g' file.txt
```

Every non-overlapping match on the line is replaced.

Extended regex

```
sed -E 's/[0-9]{1,3}(\.[0-9]{1,3}){3}/IP/g' access.log
```

With `-E`, `{}` and `()` work without backslashes (GNU ERE).

Delete matching lines

```
sed '/^#/d; /^$/d' config.txt
```

Drops comment lines and empty lines — useful for “effective config” views.

Print only rewritten matches

```
sed -n 's/.*ERROR: //p' app.log
```

`-n` suppresses default output; `p` prints only successful substitutions — like a capturing `grep`.

In-place with backup

```
sed -i.bak 's/Enable=false/Enable=true/' app.conf  
diff -u app.conf.bak app.conf
```

Inspect the diff before deleting the `.bak`.

Line addresses and ranges

```
sed -n '10,20p' file.txt          # print lines 10-20  
sed '1,5d' file.txt              # delete first five  
sed '/^## Start/,/^## End/d' doc.md
```

Addresses can be numbers, `$` (last line), or `/regex/`.

Multiple expressions

```
sed -e 's/\r$//' -e 's/[[:space:]]+$//' file.txt
```

Strip CR from CRLF files and trailing whitespace.

Use match in replacement

```
sed -E 's/(user=)[^ ]+/\1REDACTED/g' app.log
```

Parentheses capture; \1 reinserts the prefix. & is the whole match.

Change delimiter when paths have slashes

```
sed 's|/usr/local/bin|/opt/bin|g' env.sh
```

Any character after s can be the delimiter (s| |, s###).

Hold space: join two lines (advanced)

```
sed -n '1h;1!H;${x;s/\n/,/g;p}' list.txt
```

GNU sed can join lines via hold space; for heavy multi-line work prefer awk or perl.

Dry-run pattern for risky edits

```
sed 's/prod/staging/g' settings.yml | diff -u settings.yml -  
# then:  
sed -i.bak 's/prod/staging/g' settings.yml
```

Always preview against the original before -i.

Understanding Output

By default sed prints every line (after script processing) to stdout. Exit status is 0 on success; GNU sed can return non-zero for invalid options or I/O errors. In-place mode still writes a temporary file then renames — a full disk mid-edit can fail; keep backups.

Notes & Pitfalls

- Default regex is **BRE**; +, ?, | need escapes unless you use -E.
- s/a/b/ only changes the first match; forget g often.
- & in the replacement is the whole match; write \& for a literal ampersand.

- `sed` is line-oriented — multi-line HTML/XML edits are painful.
- Character classes: prefer `[:space:]` over `\s` for portable scripts.

Related Commands

- `awk` — field-oriented processing
- `grep` — filter only
- `tr` — character classes / deletion
- `perl -pe` — richer substitutions
- `diff` / `patch` — review and apply edits

Additional Resources

- `man sed`
- GNU `sed` manual

awk

Overview

awk is a pattern–action language for column-oriented text. It shines at field extraction, reports, and light transforms in pipelines — often replacing multi-step **cut/grep/sed** combinations. Prefer it when you think in **rows and fields**; use **sed** for pure stream edits and **jq** for JSON.

On Ubuntu you typically get **mawk** or **gawk** depending on package; install **gawk** when you need GNU extensions (gensub, multi-dimensional arrays, network redirections).

Syntax

```
awk [options] 'program' [file...]  
awk -f program.awk [file...]  
... | awk 'program'
```

Program form: **pattern { actions }**. Default action is **{ print }**. Built-in field vars: **\$1...\$NF**, **NF** (field count), **NR/FNR** (record numbers), **FS/OFS** (separators), **RS/ORS**, **FILENAME**.

Common Options

Option	Description
-F fs	Input field separator (same as BEGIN { FS=... })
-v var=val	Set variable before program runs
-f file	Read program from file
-W	Implementation warnings (gawk)

Key Use Cases

1. Print or reorder selected columns
2. Filter rows by field values or regex
3. Sum / average / count groups

4. Reformat logs into CSV/TSV for spreadsheets
5. Quick reports from `/etc/passwd`, `ps`, or access logs

Examples with Explanations

Print columns

```
awk '{print $1, $3}' file.txt
awk -F: '{print $1, $7}' /etc/passwd
```

Space-separated print uses `OFS` (default space). For `passwd`, `-F:` is the standard delimiter.

Filter rows

```
ps aux | awk 'NR==1 || $3+0 > 5.0'           # header + high CPU
awk -F, '$3 > 100 {print $1}' data.csv
awk '/ERROR/ && !/retry/' app.log
```

`$3+0` forces a numeric comparison (avoids string “10” < “9” surprises).

Sum and average a column

```
awk '{s+=$1} END {print s}' numbers.txt
awk '{s+=$1; n++} END {if (n) print s/n}' numbers.txt
```

`END` runs once after all input. Guard division by zero when the file may be empty.

Change separators (CSV → TSV)

```
awk -F, 'BEGIN{OFS="\t"} {print $2,$1}' data.csv
```

Set `OFS` in `BEGIN` so every `print` uses tabs.

Last field and field count

```
awk '{print $NF}' file.txt           # last field
awk '{print NF, $0}' file.txt        # how many fields per line
```

`$NF` is “last field”; `$(NF-1)` is second-to-last.

Multiple patterns / counters

```
awk '/ERROR/ {err++} /WARN/ {warn++} END {print "err=" err, "warn=" warn}' app.log
```

Patterns without braces still run the default print; here braces only tally.

Build a frequency report

```
awk -F: '{uids[$3]++} END {for (u in uids) print u, uids[u]}' /etc/passwd | sort -n
```

Associative arrays are the usual way to group. Order of `for (u in ...)` is undefined — pipe to `sort`.

Extract IP from mixed lines

```
awk 'match($0, /[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+)/ {  
    print substr($0, RSTART, RLENGTH)  
}' access.log
```

`match` sets `RSTART`/`RLENGTH` for the first match on the line (works in `mawk`/`gawk`).

Skip header, process the rest

```
awk -F, 'NR==1 {next} {sum+=$4} END {print sum}' report.csv
```

`next` jumps to the following record without running later rules on this line.

Script-friendly exit status

```
awk 'BEGIN{exit 0}' </dev/null  
# fail CI if any line exceeds threshold:  
awk -v lim=90 '$1+0 > lim {bad=1; print} END {exit bad+0}' metrics.txt
```

`END { exit code }` is useful in pipelines and checks.

Field splitting on multi-character FS (gawk)

```
gawk -F'[:,space:]*|[:,space:]*' '{print $2}' table.txt
```

When separators are messy, prefer `gawk` or preprocess with `tr/sed`.

Understanding Output

- Records are lines by default (`RS = "\n"`).
- Empty fields still count toward `NF` when `FS` is a single character (e.g. `a::b` → three fields with `-F:`).
- Uninitialized variables are `" "` or `0` depending on context (string vs numeric).
- Exit status is `0` on success; non-zero if the program calls `exit n` or the implementation hits a fatal error.

Notes & Pitfalls

- Default `FS` is **whitespace** (runs of spaces/tabs), not a single space.
- Locale can affect regex and sorting of output you pipe to `sort`.
- Complex CSV with quoted commas needs a real CSV parser (`python`, `mlr`), not naive `-F,`.
- `print $1 $2` concatenates; `print $1, $2` inserts `OFS`.
- BusyBox `awk` is smaller — stick to portable constructs in scripts meant for containers/embed.

Related Commands

- `cut` — simple fixed-field slice
- `sed` — stream edits / substitutions
- `sort` / `uniq` — often after `awk` for reports
- `jq` — JSON
- `mlr` (Miller) — richer tabular CLI if installed

Additional Resources

- `man awk`
- [GNU Awk User's Guide](#)

cut

Overview

The `cut` command extracts specific columns or fields from lines of text. It's useful for processing structured data like CSV files, logs, and delimited text.

Syntax

```
cut [options] [file...]
```

Common Options

Option	Description
<code>-f list</code>	Select fields
<code>-d char</code>	Field delimiter
<code>-c list</code>	Select characters
<code>-b list</code>	Select bytes
<code>-s</code>	Suppress lines without delimiters
<code>--complement</code>	Invert selection
<code>--output-delimiter=string</code>	Output delimiter

Field/Character Lists

Format	Description
1	Field/character 1
1,3,5	Fields 1, 3, and 5
1-5	Fields 1 through 5
1-	Field 1 to end
-5	First 5 fields
1,3-5,7	Mixed selection

Key Use Cases

1. Extract CSV columns
2. Process log files
3. Parse structured text
4. Data extraction
5. Text manipulation

Examples with Explanations

Example 1: Extract Fields

```
cut -f 1,3 -d ',' data.csv
```

Extracts fields 1 and 3 from CSV file

Example 2: Extract Characters

```
cut -c 1-10 file.txt
```

Extracts first 10 characters from each line

Example 3: Custom Delimiter

```
cut -f 2 -d ':' /etc/passwd
```

Extracts usernames from passwd file

Working with Different Delimiters

Common delimiters: - , - Comma (CSV) - : - Colon (passwd, PATH) - `\t` - Tab (TSV) - - Space
- | - Pipe

Common Usage Patterns

1. Extract usernames:

```
cut -f 1 -d ':' /etc/passwd
```

2. Get file extensions:

```
ls | cut -d '.' -f 2-
```

3. Process CSV data:

```
cut -f 2,4,6 -d ',' data.csv
```

Advanced Operations

1. Suppress delimiter-less lines:

```
cut -f 1 -d ',' -s file.csv
```

2. Change output delimiter:

```
cut -f 1,2 -d ',' --output-delimiter='|' data.csv
```

3. Complement selection:

```
cut -f 1,3 --complement -d ',' data.csv
```

Character vs Field Extraction

Character extraction (-c): - Fixed position extraction - Useful for fixed-width data - Byte-based positioning

Field extraction (-f): - Delimiter-based extraction - Variable width fields - More flexible for structured data

Performance Analysis

- Very fast operation
- Minimal memory usage
- Streaming operation
- Efficient for large files
- Good pipeline performance

Related Commands

- **awk** - More powerful field processing
- **grep** - Pattern matching
- **sed** - Stream editing
- **sort** - Sort lines

- `uniq` - Remove duplicates

Additional Resources

- [GNU cut manual](#)
- [Cut Command Examples](#)

Best Practices

1. Specify delimiters explicitly
2. Test field numbers with sample data
3. Use character extraction for fixed-width data
4. Consider using `awk` for complex operations
5. Handle missing delimiters appropriately

Common Patterns

1. Extract IP addresses:

```
cut -f 1 -d ' ' access.log
```

2. Get file sizes:

```
ls -l | cut -c 30-40
```

3. Process `PATH` variable:

```
echo $PATH | cut -f 1 -d ':'
```

Integration Examples

1. With `sort` and `uniq`:

```
cut -f 1 -d ',' data.csv | sort | uniq -c
```

2. With `grep`:

```
grep "error" log.txt | cut -f 1 -d ' '
```

3. Pipeline processing:

```
cat data.txt | cut -f 2,4 -d '|' | sort
```

Troubleshooting

1. Wrong field numbers
2. Delimiter not found
3. Character encoding issues
4. Empty fields handling
5. Multi-character delimiters (use awk instead) ## Additional Examples

```
cut -d: -f1 /etc/passwd
cut -d, -f1,3 data.csv
echo 'a b c' | cut -d' ' -f2
cut -c1-10 file.txt
```


sort

Overview

`sort` orders lines of text (or selected fields) and can merge pre-sorted files. It is a core building block for log analytics, unique counts (`sort | uniq -c`), and deterministic output in scripts. GNU `sort` (Ubuntu default) supports huge files via temporary disk, multiple keys, numeric/version/human-size modes, and stable sort.

Syntax

```
sort [OPTIONS] [FILE...]  
command | sort [OPTIONS]
```

With no files (or `-`), reads stdin. Multiple files are concatenated then sorted unless `-m` merges already-sorted inputs.

Common Options

Option	Description
<code>-n</code>	Numeric sort
<code>-h</code>	Human numeric (2K, 3M, 1G)
<code>-V</code>	Version sort (v2.10 after v2.9)
<code>-r</code>	Reverse
<code>-u</code>	Unique (drop duplicate lines after sort)
<code>-f</code>	Fold case (ignore case)
<code>-b</code>	Ignore leading blanks
<code>-k POS[,POS]</code>	Sort by key/field (see below)
<code>-t CHAR</code>	Field separator
<code>-s</code>	Stable sort (preserve order of equal keys)
<code>-R</code>	Random shuffle (hash-based)
<code>-o file</code>	Write output to file (safe for in-place with care)
<code>-m</code>	Merge already-sorted files
<code>-c / -C</code>	Check whether sorted (<code>-C</code> quiet)
<code>-z</code>	NUL-terminated lines (with <code>find -print0</code> pipelines)
<code>--parallel=N</code>	Parallel sort threads (GNU)
<code>-S size</code>	Memory buffer size (GNU)

Key syntax (-k)

Fields are 1-based. Examples: - -k2,2 — only field 2

- -k2 — field 2 through end of line
- -k2n — field 2 as number
- -k1,1 -k2,2n — first by field 1, then numeric field 2

Default field separator is non-blank to blank transitions (whitespace runs).

Examples with Explanations

Lexicographic sort

```
sort names.txt
sort -r names.txt
```

Numeric sort

```
sort -n scores.txt
printf '10\n2\n1\n' | sort -n
```

Without -n, 10 sorts before 2 as text.

Human-readable sizes

```
du -h /var/* 2>/dev/null | sort -h
du -h /var/* 2>/dev/null | sort -hr | head
```

Version numbers

```
printf 'v1.9\nv1.10\nv2.0\n' | sort -V
```

Sort by a column (whitespace)

```
# passwd-like: sort by UID (field 3) numerically
sort -t: -k3,3n /etc/passwd
```

CSV-ish second column (simple cases)

```
sort -t, -k2,2 file.csv
```

Embedded commas/quotes need a real CSV tool (`mlr`, `python`); `sort` is field-dumb.

Unique lines

```
sort -u access-hosts.txt
# equivalent pipeline often written as:
sort access-hosts.txt | uniq
```

Frequency report (`sort | uniq -c | sort`)

```
awk '{print $1}' access.log | sort | uniq -c | sort -nr | head
```

Classic “top talkers” pattern: count then re-sort by count.

Check if already sorted

```
sort -c names.txt && echo sorted
sort -C names.txt    # silent; use exit status
```

Safe “sort in place”

```
sort -o names.txt names.txt
```

GNU sort handles `-o` same-file correctly; `sort file > file` truncates first and destroys data.

Merge pre-sorted shards

```
sort -m part-a.txt part-b.txt part-c.txt > all-sorted.txt
```

Much faster than re-sorting the concatenation when inputs are already ordered.

Stable sort by key, preserve secondary order

```
sort -s -k1,1 events.log
```

NUL-safe with find

```
find . -type f -print0 | sort -z | xargs -0 ls -l
```

Randomize lines

```
sort -R items.txt | head -n 5
```

For cryptographic shuffles use `shuf` instead of relying on `sort -R` alone.

Understanding Output

Output is the full lines in sorted order. With `-u`, only the first of equal lines (per current key/order rules) remains. Exit status 0 on success; `sort -c` returns non-zero if out of order.

Locale (LC_COLLATE / LANG) affects byte vs dictionary order. For scripts that need byte-wise sort:

```
LC_ALL=C sort file
```

Notes & Pitfalls

- **Locale surprises:** `en_US.UTF-8` vs `C` can reorder punctuation and case. Pin `LC_ALL=C` for deterministic machine output.
- Numeric sort of non-numeric prefixes: `sort -n` parses a leading number; garbage may sort as zero.
- `-k2` vs `-k2,2` is a frequent bug — open-ended keys include the rest of the line.
- Large sorts use `/tmp` (or `$TMPDIR`); disk-full mid-sort fails — free space or set `-S/TMPDIR` on big data hosts.
- `sort -u` uniqueness respects the sort keys/options, not always the whole line if keys are restricted — know what you keyed.
- Prefer `shuf` for sampling random lines; prefer `mlr/csvkit` for real CSV.

Related Commands

- `uniq` — adjacent unique/count (usually after `sort`)
- `shuf` — random permutations
- `comm` — compare two sorted files
- `join` — relational join on sorted files
- `awk` / `cut` — extract fields before sorting
- `wc -l` — count lines after filtering

Additional Resources

- `man sort`
- GNU coreutils info: `info sort`

uniq

Overview

`uniq` reports or filters **adjacent** duplicate lines. It does not search the whole file for duplicates unless identical lines are already next to each other — which is why the standard pattern is `sort | uniq`. GNU `uniq` can count runs, show only duplicates, or skip comparison of leading fields/characters.

Syntax

```
uniq [OPTIONS] [INPUT [OUTPUT]]
command | uniq [OPTIONS]
```

Common Options

Option	Description
-c	Prefix each line with occurrence count
-d	Only print lines that are duplicated
-D	Print all duplicate lines (GNU; every copy in a run)
-u	Only print lines that are unique (appear once)
-i	Case-insensitive compare
-f N	Skip first N fields when comparing
-s N	Skip first N characters when comparing
-w N	Compare at most N characters
-z	NUL-terminated lines (GNU)

Examples with Explanations

Remove adjacent duplicates

```
printf 'a\na\nb\na\n' | uniq
# → a / b / a    (the second 'a' group is separate)
```

Without sorting, non-adjacent duplicates remain.

The usual “unique lines in file” pattern

```
sort names.txt | uniq
sort -u names.txt           # often faster one-process equivalent
```

Count occurrences

```
sort access-hosts.txt | uniq -c
sort access-hosts.txt | uniq -c | sort -nr | head
```

`uniq -c` counts **runs**; sorting first groups equal lines so the count is global.

Top IPs from a log (field extract → sort → uniq -c → sort)

```
awk '{print $1}' /var/log/nginx/access.log \
| sort \
| uniq -c \
| sort -nr \
| head -n 20
```

Only lines that appear more than once

```
sort ids.txt | uniq -d
```

Only lines that appear exactly once

```
sort ids.txt | uniq -u
```

Case-insensitive uniqueness

```
printf 'Foo\nfoo\nBAR\n' | sort -f | uniq -i
```

Sort with matching case rules (`-f`) so case variants become adjacent.

Ignore leading fields when comparing

```
# two columns: timestamp id - unique by id only among adjacent
sort -k2,2 events.txt | uniq -f 1
```

-f 1 skips field 1 (the timestamp) during comparison; whitespace field rules match `sort`'s default.

Compare only a prefix of each line

```
uniq -w 8 hashes.txt
```

Useful when a fixed-width key leads the line.

Show every line that is part of a duplicate group

```
sort data.txt | uniq -D
```

NUL-separated path lists

```
find . -type f -print0 | sort -z | uniq -z | xargs -0 ...
```

Rare but correct for exotic pipelines that stay NUL-safe end-to-end.

Count total unique lines (quick)

```
sort file.txt | uniq | wc -l
# or
sort -u file.txt | wc -l
```

Understanding Output

Default: one line per adjacent run of identical lines. With `-c`, output looks like:

```
12 10.0.0.5
 7 10.0.0.8
 1 10.0.0.9
```

Counts are right-aligned in a fixed width (GNU). Leading spaces are normal — strip with `awk '{print $1}'` when you need the number alone.

Notes & Pitfalls

- **uniq without sort only collapses neighbors.** This is the #1 misconception.
- `sort -u` and `sort | uniq` are often equivalent for whole-line uniqueness; prefer one clear form in scripts.
- Locale can affect whether lines compare equal; use `LC_ALL=C` for byte-wise behavior in tooling.
- `uniq -c | sort -n` sorts by the whole line as text unless you use `sort -nr` and account for leading spaces — `sort -k1,1nr` is clearer on counted output.
- Blank lines are real lines; consecutive empties collapse under `uniq`.
- For multisets and complex keys, `awk` associative arrays or `datamash` may be clearer than `uniq -f` gymnastics.

Related Commands

- `sort / sort -u` — order and unique
- `comm` — compare two sorted unique streams
- `awk '!seen[$0]++'` — unique without sorting (memory-bound, order-preserving first wins)
- `grep -v` — filter patterns, not duplicate runs
- `wc -l` — count remaining lines

Additional Resources

- `man uniq`
- GNU coreutils info: `info uniq`

tr

Overview

The **tr** (translate) command translates or deletes characters from standard input. It's used for character substitution, deletion, and squeezing repeated characters.

Syntax

```
tr [options] set1 [set2]
```

Common Options

Option	Description
-c	Complement set1
-d	Delete characters in set1
-s	Squeeze repeated characters
-t	Truncate set1 to length of set2

Character Sets

Set	Description
[:alnum:]	Alphanumeric characters
[:alpha:]	Alphabetic characters
[:digit:]	Digits 0-9
[:lower:]	Lowercase letters
[:upper:]	Uppercase letters
[:space:]	Whitespace characters
[:punct:]	Punctuation characters
[:print:]	Printable characters
[:cntrl:]	Control characters

Key Use Cases

1. Case conversion
2. Character replacement
3. Delete unwanted characters
4. Format text data
5. Clean input data

Examples with Explanations

Example 1: Uppercase Conversion

```
echo "hello world" | tr '[:lower:]' '[:upper:]'
```

Output: HELLO WORLD

Example 2: Delete Characters

```
echo "hello123world" | tr -d '[:digit:]'
```

Output: helloworld

Example 3: Replace Characters

```
echo "hello world" | tr ' ' '_'
```

Output: hello_world

Example 4: Squeeze Repeated Characters

```
echo "hello    world" | tr -s ' '
```

Output: hello world

Common Usage Patterns

1. Convert to lowercase:

```
echo "HELLO" | tr '[:upper:]' '[:lower:]'
```

2. Remove newlines:

```
cat file.txt | tr -d '\n'
```

3. Replace multiple characters:

```
echo "a,b;c:d" | tr ',;:' ' ' '
```

Character Ranges

1. Letter ranges:

```
echo "hello" | tr 'a-z' 'A-Z'
```

2. Number ranges:

```
echo "123" | tr '1-3' 'abc'
```

3. Custom ranges:

```
echo "hello" | tr 'helo' '1234'
```

Advanced Usage

1. Complement sets:

```
echo "hello123" | tr -cd '[:alpha:]' # Keep only letters
```

2. Multiple operations:

```
echo "Hello World" | tr '[:upper:]' '[:lower:]' | tr ' ' '_'
```

3. ROT13 encoding:

```
echo "hello" | tr 'a-zA-Z' 'n-zA-mN-ZA-M'
```

Text Processing

1. Clean CSV data:

```
cat data.csv | tr -d '"' | tr ', ' '\t'
```

2. Format phone numbers:

```
echo "1234567890" | tr '0-9' '(###) ###-####'
```

3. Remove control characters:

```
cat file.txt | tr -d '[:cntrl:]'
```

Performance Analysis

- Very fast character processing
- Stream-based operation
- Minimal memory usage
- Efficient for large files
- Good pipeline performance

Related Commands

- `sed` - Stream editor
- `awk` - Text processing
- `cut` - Extract fields
- `sort` - Sort lines
- `uniq` - Remove duplicates

Best Practices

1. Use character classes for portability
2. Test transformations on sample data
3. Combine with other text tools
4. Handle special characters carefully
5. Consider locale settings

Data Cleaning

1. Remove punctuation:

```
echo "Hello, World!" | tr -d '[:punct:]'
```

2. Normalize whitespace:

```
echo "hello    world" | tr -s '[:space:]' ' '
```

3. Extract numbers:

```
echo "abc123def456" | tr -cd '[:digit:]'
```

File Processing

1. Convert line endings:

```
tr -d '\r' < dos_file.txt > unix_file.txt
```

2. Create word list:

```
cat text.txt | tr '[:space:][:punct:]' '\n' | tr -s '\n'
```

3. Count characters:

```
cat file.txt | tr -cd '[:alpha:]' | wc -c
```

Integration Examples

1. With find for filename processing:

```
find . -name "*.txt" | tr '[:upper:]' '[:lower:]'
```

2. Log processing:

```
tail -f access.log | tr ',' '\t' | cut -f1
```

3. Data format conversion:

```
cat data.txt | tr ';' ', ' > data.csv
```

Scripting Applications

1. Input validation:

```
validate_input() {
    echo "$1" | tr -cd '[:alnum:]' | grep -q . || return 1
}
```

2. Password generation:

```
generate_password() {
    tr -dc 'A-Za-z0-9' < /dev/urandom | head -c 12
}
```

Special Characters

1. Handle tabs:

```
echo -e "hello\tworld" | tr '\t' ' '
```

2. Process escape sequences:

```
echo "hello\nworld" | tr '\\\n' '/'
```

3. Unicode handling:

```
echo "café" | tr 'é' 'e'
```

Troubleshooting

1. Character encoding issues
2. Locale-specific behavior
3. Special character handling
4. Set length mismatches
5. Unexpected transformations

Security Applications

1. Sanitize input:

```
echo "$user_input" | tr -cd '[:alnum:]._-'
```

2. Remove dangerous characters:

```
echo "$filename" | tr -d '/<>:*?"\\'
```

Performance Optimization

1. Use character classes:

```
# Faster
tr '[:lower:]' '[:upper:]'
# Slower
tr 'abcdefghijklmnopqrstuvwxyz' 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

2. Combine operations:

```
# Single tr call is faster
echo "Hello World" | tr '[:upper:]' '[:lower:]_'
```

Real-world Examples

1. Log analysis:

```
grep ERROR /var/log/app.log | tr '[:upper:]' '[:lower:]' | sort | uniq -c
```

2. Data migration:

```
cat old_format.txt | tr '|' ', ' | tr -s ' ' > new_format.csv
```

3. Text normalization:

```
cat document.txt | tr -s '[:space:]' ' ' | tr '[:upper:]' '[:lower:]'
```


head

Overview

The **head** command displays the first lines of files or input streams. By default, it shows the first 10 lines, making it useful for previewing file contents.

Syntax

```
head [options] [file...]
```

Common Options

Option	Description
-n num	Show first num lines
-c num	Show first num bytes
-q	Suppress headers
-v	Always show headers
-z	Line delimiter is NUL
--lines=num	Same as -n
--bytes=num	Same as -c

Key Use Cases

1. Preview file contents
2. Extract file headers
3. Sample data examination
4. Log file monitoring
5. Quick file inspection

Examples with Explanations

Example 1: Default Usage

```
head file.txt
```

Shows first 10 lines of file

Example 2: Specific Line Count

```
head -n 5 file.txt
```

Shows first 5 lines

Example 3: Multiple Files

```
head -n 3 *.txt
```

Shows first 3 lines of each txt file

Example 4: Byte Count

```
head -c 100 file.txt
```

Shows first 100 bytes

Common Usage Patterns

1. Quick file preview:

```
head -20 logfile.log
```

2. CSV header inspection:

```
head -1 data.csv
```

3. Combined with tail:

```
head -50 file.txt | tail -10
```

Advanced Usage

1. Suppress filename headers:

```
head -q file1.txt file2.txt
```

2. Always show headers:

```
head -v file.txt
```

3. Process substitution:

```
head -5 <(command)
```

Performance Analysis

- Very fast for small line counts
- Efficient streaming operation
- Minimal memory usage
- Good for large files
- Stops reading after required lines

Related Commands

- `tail` - Show last lines
- `more` - Page through files
- `less` - Advanced pager
- `cat` - Display entire file
- `sed` - Stream editor

Best Practices

1. Use appropriate line counts
2. Combine with other text tools
3. Consider byte vs line counting
4. Use for quick file validation
5. Helpful for debugging scripts

Integration Examples

1. Log analysis:

```
head -100 /var/log/syslog | grep error
```

2. Data sampling:

```
head -1000 large_dataset.csv > sample.csv
```

3. Script debugging:

```
head -5 "$input_file" | while read line; do
    echo "Processing: $line"
done
```

Scripting Applications

1. File validation:

```
if head -1 "$file" | grep -q "^#"; then
    echo "File has header"
fi
```

2. Quick content check:

```
head -n 1 *.conf | grep -v "^#"
```

Additional Examples

```
head file.txt
head -n 50 file.txt
head -n -20 file.txt          # all but last 20 (GNU)
head -c 1K binary.dat
```

tail

Overview

The **tail** command displays the last lines of files or input streams. It's essential for monitoring log files and examining file endings.

Syntax

```
tail [options] [file...]
```

Common Options

Option	Description
-n num	Show last num lines
-c num	Show last num bytes
-f	Follow file changes
-F	Follow with retry
-q	Suppress headers
-v	Always show headers
-s num	Sleep seconds between checks
--pid=pid	Terminate after process dies
--retry	Keep trying to open file

Key Use Cases

1. Monitor log files
2. View file endings
3. Real-time file watching
4. Debug running processes
5. System monitoring

Examples with Explanations

Example 1: Default Usage

```
tail file.txt
```

Shows last 10 lines of file

Example 2: Follow Log File

```
tail -f /var/log/syslog
```

Continuously monitors log file for new entries

Example 3: Specific Line Count

```
tail -n 20 error.log
```

Shows last 20 lines

Example 4: Multiple Files

```
tail -f /var/log/*.log
```

Follows multiple log files simultaneously

Follow Mode Options

Option	Behavior
-f	Follow by file descriptor
-F	Follow by name (handles rotation)
--retry	Keep trying if file doesn't exist
--pid=pid	Stop when process dies

Common Usage Patterns

1. Monitor application logs:

```
tail -f /var/log/apache2/error.log
```

2. Watch system logs:

```
tail -f /var/log/messages
```

3. Debug scripts:

```
tail -f script.log &  
./script.sh
```

Advanced Usage

1. Follow with retry:

```
tail -F /var/log/app.log
```

2. Stop after process ends:

```
tail -f --pid=$PID logfile.log
```

3. Custom sleep interval:

```
tail -f -s 0.1 fast_changing.log
```

Performance Analysis

- Efficient for file monitoring
- Low CPU usage in follow mode
- Handles log rotation well with -F
- Good for real-time monitoring
- Minimal memory footprint

Related Commands

- `head` - Show first lines
- `less` - Advanced pager
- `watch` - Execute commands periodically
- `journalctl` - Systemd log viewer

- `multitail` - Monitor multiple files

Best Practices

1. Use `-F` for log files that rotate
2. Combine with `grep` for filtering
3. Use `-pid` for temporary monitoring
4. Consider `multitail` for multiple files
5. Be aware of file descriptor limits

Integration Examples

1. Filtered monitoring:

```
tail -f /var/log/syslog | grep ERROR
```

2. Multiple log analysis:

```
tail -f /var/log/{messages,secure,maillog}
```

3. Application debugging:

```
tail -f app.log | while read line; do  
    echo "$(date): $line"  
done
```

Log Rotation Handling

1. Follow by name (handles rotation):

```
tail -F /var/log/app.log
```

2. Retry if file disappears:

```
tail -f --retry /var/log/app.log
```

Scripting Applications

1. Wait for log entry:

```
tail -f app.log | grep -q "Server started" && echo "Ready"
```


2. Monitor until condition:

```
timeout 60 tail -f deploy.log | grep -q "Deployment complete"
```

System Monitoring

1. Watch system load:

```
tail -f /proc/loadavg
```

2. Monitor memory:

```
watch -n 1 'tail -5 /proc/meminfo'
```

Additional Examples

```
tail file.txt
tail -n 100 file.txt
tail -f /var/log/syslog      # follow
tail -F app.log             # follow with rename reopen
journalctl -u nginx -f      # often better for units
```

WC

Overview

The `wc` (word count) command counts lines, words, characters, and bytes in files or input streams. It's essential for text analysis and file statistics.

Syntax

```
wc [options] [file...]
```

Common Options

Option	Description
<code>-l</code>	Count lines only
<code>-w</code>	Count words only
<code>-c</code>	Count bytes only
<code>-m</code>	Count characters only
<code>-L</code>	Length of longest line
<code>--files0-from=file</code>	Read null-separated filenames

Default Output Format

Without options, `wc` shows:

```
lines words bytes filename
```

Example output:

```
42  156  892 file.txt
```

Key Use Cases

1. Count lines in files
2. Analyze text statistics
3. Monitor file growth
4. Validate data processing
5. Script automation

Examples with Explanations

Example 1: Basic Count

```
wc file.txt
```

Shows lines, words, and bytes count

Example 2: Lines Only

```
wc -l file.txt
```

Shows only line count

Example 3: Multiple Files

```
wc *.txt
```

Shows counts for all text files plus totals

Understanding Counts

- **Lines:** Number of newline characters
- **Words:** Sequences of non-whitespace characters
- **Characters:** Including multibyte characters
- **Bytes:** Raw byte count (may differ from characters)

Common Usage Patterns

1. Count log entries:

```
wc -l /var/log/syslog
```

2. Monitor file growth:

```
watch "wc -l growing_file.log"
```

3. Pipeline counting:

```
ps aux | wc -l
```

Advanced Usage

1. Longest line length:

```
wc -L file.txt
```

2. Character vs byte count:

```
wc -m file.txt # characters  
wc -c file.txt # bytes
```

3. Multiple file totals:

```
wc -l *.log
```

Pipeline Integration

1. Count command output:

```
ls | wc -l
```

2. Count unique lines:

```
sort file.txt | uniq | wc -l
```

3. Count pattern matches:

```
grep "error" log.txt | wc -l
```

Performance Analysis

- Very fast operation
- Efficient for large files
- Minimal memory usage
- Good pipeline performance
- Streaming capability

Related Commands

- `nl` - Number lines
- `sort` - Sort lines
- `uniq` - Remove duplicates
- `cut` - Extract fields
- `awk` - Text processing

Additional Resources

- [GNU wc manual](#)
- [Word Count Examples](#)

Best Practices

1. Use specific options for clarity
2. Combine with other text tools
3. Consider character encoding
4. Use in scripts for validation
5. Monitor with watch for real-time updates

Scripting Examples

1. File size validation:

```
if [ $(wc -l < file.txt) -gt 1000 ]; then
    echo "File too large"
fi
```

2. Progress monitoring:

```
TOTAL=$(wc -l < input.txt)
echo "Processing $TOTAL lines"
```

3. Log rotation trigger:

```
[ $(wc -l < logfile) -gt 10000 ] && logrotate config
```

Character Encoding

Difference between `-c` and `-m`: - `-c` counts bytes - `-m` counts characters (important for UTF-8)

Example with Unicode:

```
echo "café" | wc -c # 5 bytes
echo "café" | wc -m # 4 characters
```

Common Patterns

1. Count non-empty lines:

```
grep -c "." file.txt
```

2. Count files in directory:

```
ls -l | wc -l
```

3. Count unique users:

```
cut -d: -f1 /etc/passwd | wc -l
```

Integration Examples

1. With find:

```
find . -name "*.py" -exec wc -l {} + | tail -1
```

2. With xargs:

```
find . -name "*.txt" | xargs wc -l
```

3. Log analysis:

```
tail -f access.log | while read line; do
    echo "Total requests: $(wc -l < access.log)"
done
```

Troubleshooting

1. Binary files giving unexpected results
2. Character encoding issues
3. Very large files
4. Empty files
5. Permission problems

Real-world Applications

1. Code metrics:

```
find . -name "*.py" | xargs wc -l | tail -1
```

2. Data validation:

```
[ $(wc -l < data.csv) -eq $(wc -l < expected.csv) ]
```

3. Monitoring:

```
wc -l /var/log/messages | awk '{print $1}' > line_count.txt
```

tee

Overview

`tee` reads stdin and writes it to both standard output **and** one or more files. It is the standard way to keep a pipeline's live output while saving a copy for logs, audits, or later processing. Named after a T-splitter in plumbing.

Syntax

```
command | tee [OPTIONS] [FILE...]  
command | tee [OPTIONS] FILE | other-command
```

With no files, `tee` only copies stdin to stdout (still useful with options like `-a` rarely alone).

Common Options

Option	Description
<code>-a</code> / <code>--append</code>	Append to files instead of overwriting
<code>-i</code> / <code>--ignore-interrupts</code>	Ignore SIGINT (less common in scripts)
<code>-p</code>	Diagnose errors writing to non-pipes (GNU; behavior with SIGPIPE)

Safety

- `tee file` **truncates** `file` by default when the pipeline starts — same class of footgun as `> file`. Use `-a` to append.
- Writing into system paths needs appropriate privileges (`sudo tee` pattern below).
- Do not `tee` secrets into world-readable locations; check `umask` and directory permissions.

Examples with Explanations

Save output and still see it

```
ping -c 4 1.1.1.1 | tee ping.log
```

Append to a running log

```
./deploy.sh 2>&1 | tee -a deploy.log
```

2>&1 captures stderr too so the log is complete.

Split to multiple files

```
./build.sh 2>&1 | tee build.log build-copy.log
```

Continue the pipeline after saving

```
dmesg | tee dmesg.full | grep -i error
```

Full capture in dmesg.full; only errors scroll on the terminal.

Write files that need root (classic sudo tee)

```
echo 'net.ipv4.ip_forward=1' | sudo tee /etc/sysctl.d/99-forward.conf
```

Why not `sudo echo ... > file`? The redirect is applied by the **local shell** before `sudo`, so the file open happens as you, not root. `sudo tee` opens the file as root.

Same pattern: append as root

```
echo 'export PATH="/opt/bin:$PATH"' | sudo tee -a /etc/environment
```

(Validate distro conventions before editing `/etc/environment`.)

Suppress terminal output but keep the file

```
make 2>&1 | tee build.log >/dev/null
# or simply:
make 2>&1 > build.log
```

`tee` is only needed when you still want a branch (multiple files, or later pipe stages).

Capture and process with jq later

```
curl -sS https://api.github.com/repos/jqlang/jq \
| tee repo.json \
| jq -r '.stargazers_count'
```

Timestamped session log with a second process

```
./long-job.sh 2>&1 | tee -a "job-$(date +%F).log"
```

Combine with process substitution (bash) for dual processing

```
./app 2>&1 | tee >(logger -t myapp) | grep -i error
```

Sends a copy to syslog via `logger` while grepping errors on the terminal (bash/zsh).

Preserve exit status of the left side (bash pipefail)

```
set -o pipefail
make 2>&1 | tee build.log
```

Without `pipefail`, the pipeline's exit status is `tee`'s (usually 0) even if `make` failed.

Understanding Output

Whatever arrives on `stdin` is duplicated to each listed file and to `stdout`, byte-for-byte (text or binary). Order is preserved. If a file cannot be opened, GNU `tee` reports an error and still attempts other outputs; check exit status in scripts.

Notes & Pitfalls

- **Truncation default:** `tee log` overwrites; `tee -a log` appends.
- **Exit codes:** with default bash pipes, failures on the left can be masked — enable `set -o pipefail` in scripts.
- **Buffering:** when stdout is a pipe, some programs fully buffer; you may see delayed terminal output. Tools like `stdbuf -oL` can force line buffering when needed.
- Binary data works; avoid viewing huge binary tees in the terminal.
- `tee` is not a remote multipath tool — for network fan-out use `socat`, message buses, or log shippers.
- Multiple writers appending to one log without coordination can interleave lines; for concurrent jobs prefer separate logs or a proper logger.

Related Commands

- shell redirects `>` / `>>` — single destination, no T-split
- `script` — record an entire interactive terminal session
- `logger` — send lines to syslog
- `sponge` (moreutils) — soak stdin then write (safe in-place pipeline edits)
- `pv` — progress meter in pipelines (if installed)

Additional Resources

- `man tee`
- GNU coreutils info: `info tee`

xargs

Overview

`xargs` builds and executes command lines from standard input. It bridges tools that emit lists (files, IDs, URLs) to commands that expect `argv` parameters. Essential in pipelines with `find`, `git`, and text filters — and easy to get wrong with spaces or special characters unless you use NUL-delimited mode.

Syntax

```
command | xargs [options] [utility [args...]]
xargs [options] -a file [utility [args...]]
```

If utility is omitted, `xargs` defaults to `/bin/echo`.

Common Options

Option	Description
<code>-0 / --null</code>	Input items are NUL-separated (safe for arbitrary filenames)
<code>-n N</code>	At most N args per command invocation
<code>-n 1</code>	One arg per invocation
<code>-P N</code>	Run up to N processes in parallel
<code>-I {}</code>	Replace {} in the command template with each item
<code>-i</code>	Like <code>-I {}</code> (deprecated form on some systems)
<code>-t</code>	Print the command before running it
<code>-p</code>	Prompt before each command
<code>-r</code>	Do not run if stdin is empty (GNU)
<code>-d delim</code>	Input delimiter (GNU)
<code>-L N</code>	Use at most N lines per command
<code>-a file</code>	Read items from file instead of stdin
<code>-x</code>	Exit if a single item is too long for the command line

Safety

- **Never** pipe raw `find` output into bare `xargs` when names may contain spaces or newlines. Use `find ... -print0 | xargs -0 ...`.
- `xargs` splits on whitespace by default — a filename `my file.txt` becomes two arguments.
- Parallel `-P` can overwhelm disks, APIs, or remote hosts; start low.
- Interactive prompts (`-p`) are for humans; do not use in unattended scripts without care.

Examples with Explanations

Basic: echo words as args

```
echo 'one two three' | xargs
# runs: echo one two three
```

Delete files listed by find (NUL-safe)

```
find /tmp -name '*.tmp' -print0 | xargs -0 rm -f
```

Equivalent modern find:

```
find /tmp -name '*.tmp' -delete
```

One invocation per item with placeholder

```
find . -name '*.md' -print0 | xargs -0 -I {} cp {} /backup/docs/
```

`-I {}` forces one item at a time and allows the placeholder anywhere in the command.

Limit batch size

```
cat ids.txt | xargs -n 50 echo
# or feed a real tool:
cat ids.txt | xargs -n 100 my-bulk-api-delete
```

Avoids “argument list too long” and huge single CLI lines.

Parallel compression

```
find . -type f -name '*.log' -print0 \  
  | xargs -0 -P 4 -n 1 gzip
```

Four gzip workers, one file each.

Show commands (-t) while learning

```
printf 'a\nb\nc\n' | xargs -n 1 -t rm -f
```

Skip empty input (GNU)

```
grep -rl 'obsolete' . | xargs -r rm  
# better NUL-safe:  
grep -rlZ 'obsolete' . | xargs -0 -r rm
```

Without `-r`, some xargs still run the command once with no args — dangerous for `rm`.

Lines as items (GNU -d)

```
xargs -d '\n' -n 1 echo < urls.txt
```

Still fails if lines contain embedded newlines (rare); for filenames prefer NUL.

Placeholders with multiple uses

```
cat hosts.txt | xargs -I% sh -c 'ssh % "hostname; uptime"'
```

Quote carefully; prefer a real loop for complex remote commands.

From a file directly

```
xargs -a packages.txt -n 1 sudo apt-get install -y
```

(Still review package lists before mass installs.)

Combine with git

```
git ls-files -z '*.png' | xargs -0 ls -l
git grep -z -l 'TODO' | xargs -0 "$EDITOR"
```

Understanding Output

`xargs` itself is quiet unless `-t/-p` is set; you see the **utility’s** stdout/stderr. Exit status is non-zero if any invocation fails (details vary; with `-P`, failures can interleave). Use `-t` when a pipeline “does nothing” — often the built command is not what you expected.

Notes & Pitfalls

- NUL safety is the professional default for file paths: `-print0 / -z / -Z` upstream + `xargs -0`.
- `-I {}` implies `-n 1` and disables some batching; good for clarity, slower for huge sets.
- Quoting inside `sh -c` with `xargs` is a common injection footgun — validate input or avoid `sh -c` when possible.
- “Argument list too long” (E2BIG) is exactly what batching with `-n` fixes.
- Busybox or non-GNU `xargs` may lack `-0`, `-r`, `-a`, `-d`, or `-P`. On Ubuntu GNU `findutils` `xargs`, these are available.
- For complex per-file logic, a `while IFS= read -r -d ' ' f; do ...; done` loop is often clearer than nested `xargs`.

Related Commands

- `find -exec ... {} +` — batch without a separate `xargs`
- `parallel` (GNU) — richer job control if installed
- `xargs -0 + grep -Z / git -z` — NUL-safe ecosystems
- `printf / tr '\n' '\0'` — convert delimiters carefully
- shell `while read` loops — when you need full shell features

Additional Resources

- `man xargs`
- `man find` (especially `-print0`, `-exec`)

diff

Overview

The `diff` command compares files line by line and displays the differences. It's essential for version control, code review, and file comparison tasks.

Syntax

```
diff [options] file1 file2
diff [options] directory1 directory2
```

Common Options

Option	Description
<code>-u</code>	Unified diff format
<code>-c</code>	Context diff format
<code>-i</code>	Ignore case differences
<code>-w</code>	Ignore whitespace
<code>-b</code>	Ignore changes in whitespace
<code>-B</code>	Ignore blank lines
<code>-r</code>	Recursive directory comparison
<code>-q</code>	Brief output (only if files differ)
<code>-s</code>	Report identical files
<code>-y</code>	Side-by-side comparison
<code>--color</code>	Colorize output

Output Formats

Format	Description
Normal	Default format with line numbers
Unified (-u)	Git-style format
Context (-c)	Shows context around changes
Side-by-side (-y)	Two-column comparison

Format	Description
--------	-------------

Key Use Cases

1. Compare file versions
2. Code review
3. Configuration changes
4. Backup verification
5. Patch creation

Examples with Explanations

Example 1: Basic Comparison

```
diff file1.txt file2.txt
```

Shows differences between two files

Example 2: Unified Format

```
diff -u original.txt modified.txt
```

Shows differences in unified format (like Git)

Example 3: Directory Comparison

```
diff -r dir1/ dir2/
```

Recursively compares two directories

Understanding Output

Normal format symbols: - a - Added lines - d - Deleted lines - c - Changed lines - < - Lines from first file - > - Lines from second file

Example:

```
2c2
< old line
---
> new line
```

Common Usage Patterns

1. Ignore whitespace:

```
diff -w file1.txt file2.txt
```

2. Side-by-side view:

```
diff -y file1.txt file2.txt
```

3. Quick check if files differ:

```
diff -q file1.txt file2.txt
```

Advanced Options

Option	Description
<code>--exclude=pattern</code>	Exclude files matching pattern
<code>--exclude-from=file</code>	Exclude patterns from file
<code>-x pattern</code>	Exclude files matching pattern
<code>-N</code>	Treat absent files as empty
<code>-a</code>	Treat all files as text
<code>--strip-trailing-cr</code>	Strip carriage returns

Patch Creation

Create patches for later application:

```
diff -u original.txt modified.txt > changes.patch
```

Apply patches:

```
patch original.txt < changes.patch
```

Performance Analysis

- Efficient for text files
- Memory usage scales with file size
- Good for moderate-sized files
- Consider alternatives for binary files
- Fast for small differences

Related Commands

- `cmp` - Compare files byte by byte
- `comm` - Compare sorted files
- `patch` - Apply diff patches
- `git diff` - Git version comparison
- `vimdiff` - Visual diff editor

Additional Resources

- [GNU diff manual](#)
- [Diff Command Guide](#)

Best Practices

1. Use unified format for patches
2. Ignore irrelevant whitespace
3. Use recursive mode for directories
4. Consider binary file handling
5. Use with version control systems

Directory Comparison

1. Compare structures:

```
diff -r --brief dir1/ dir2/
```

2. Exclude files:

```
diff -r --exclude="*.log" dir1/ dir2/
```

3. Show only differences:

```
diff -r -q dir1/ dir2/
```

Integration Examples

1. With git:

```
git diff > changes.patch
```

2. With find:

```
diff <(find dir1 -type f | sort) <(find dir2 -type f | sort)
```

3. Configuration management:

```
diff -u /etc/config.orig /etc/config
```

Scripting Applications

1. Backup verification:

```
if diff -q original.txt backup.txt > /dev/null; then
    echo "Backup is identical"
fi
```

2. Configuration monitoring:

```
diff /etc/passwd /etc/passwd.bak || echo "Password file changed"
```

3. Automated testing:

```
diff expected_output.txt actual_output.txt || exit 1
```

Special Cases

1. Binary files:

```
diff -q binary1 binary2
```

2. Large files:

```
diff --speed-large-files file1 file2
```

3. Case-insensitive:

```
diff -i file1.txt file2.txt
```

Troubleshooting

1. Binary file warnings
2. Memory issues with large files
3. Character encoding problems

4. Permission denied errors
5. Directory structure differences

Output Redirection

1. Save differences:

```
diff file1.txt file2.txt > differences.txt
```

2. Suppress output:

```
diff file1.txt file2.txt > /dev/null
```

3. Error handling:

```
diff file1.txt file2.txt 2>&1 | tee diff.log
```

Color Output

Modern diff versions support color:

```
diff --color=always file1.txt file2.txt
```

Environment variable:

```
export DIFF_COLORS="old=31:new=32:hunk=36"
```

comm

Overview

`comm` compares **two sorted files** line by line and outputs three columns: lines only in file1, only in file2, and common to both. Perfect for set operations on sorted word lists, user lists, and package inventories. Unsorted input produces garbage — sort first.

Syntax

```
comm [options] file1 file2
```

Common Options

Option	Description
-1	Suppress column 1 (unique to file1)
-2	Suppress column 2 (unique to file2)
-3	Suppress column 3 (common)
-z	NUL-terminated lines
--check-order	Check that inputs are sorted
--nocheck-order	Skip order check
--output-delimiter=STR	Column separator
--total	Output summary counts (newer)

Examples with Explanations

Basic three columns

```
sort a.txt -o a.s
sort b.txt -o b.s
comm a.s b.s
```

Only in first / second / both

```
comm -23 a.s b.s      # only in a
comm -13 a.s b.s      # only in b
comm -12 a.s b.s      # common
```

Mnemonic: suppress the columns you don't want (-23 keeps col1).

Users differences

```
getent passwd | cut -d: -f1 | sort > host_users
# compare to expected list
comm -23 expected_users host_users    # expected but missing
comm -13 expected_users host_users    # unexpected present
```

Package set ops

```
rpm -qa | sort > now
comm -13 baseline now    # newly installed vs baseline
```

Check order

```
comm --check-order a.txt b.txt
```

Process substitution

```
comm -12 <(sort list1) <(sort list2)
```

Empty markers

Columns are tab-separated; empty leading fields mean a line isn't in earlier columns. Use `cat -A` to see tabs:

```
comm a.s b.s | cat -A
```

Notes / Pitfalls

- **Both files must be sorted** in the same collation (`LC_ALL=C sort` for stable byte order).
- Locale sort order can surprise — set `LC_ALL=C` for machine lists.
- Duplicate lines: `comm` is multiset-aware in the sense of sorted runs; unique with `sort -u` if needed.
- Binary files / CRLF: normalize first (`dos2unix`).
- For unsorted fuzzy joins, use `join`, `grep -F -f`, or databases.

2026-relevant notes

- Still excellent for offline set diffs without loading Python.
- Large inventories: ensure `sort` can use tmp space (`TMPDIR`).
- Combine with `rg --files` lists for workspace membership diffs.

Related Commands

- `sort` / `uniq` — prepare inputs
- `join` — join on fields
- `diff` / `git diff` — line-oriented diffs with context
- `grep -F -f` — filter by list
- `cmp` — binary compare

Additional Resources

- `man comm`

paste

Overview

`paste` merges lines of files **side by side**, writing corresponding lines separated by tabs (or another delimiter). The counterpart mindset to `cut` (split columns) — `paste` builds columns from separate files or serializes lines with `-s`.

Syntax

```
paste [options] [file...]
```

Common Options

Option	Description
<code>-d LIST, --delimiters=LIST</code>	Cycle delimiter characters instead of tab
<code>-s, --serial</code>	Paste one file at a time (serialize lines of each file)
<code>-z, --zero-terminated</code>	NUL-terminated lines

Use `-` for stdin.

Examples with Explanations

Side by side

```
paste names.txt ages.txt
paste <(echo -e 'a\nb') <(echo -e '1\n2')
```

Custom delimiter

```
paste -d, names.txt ages.txt
paste -d'|' a.txt b.txt c.txt
paste -d '\t|' a.txt b.txt      # cycle delimiters
```

Serialize lines of one file

```
# join all lines with comma
paste -sd, file.txt
# words to CSV row
paste -sd' ' words.txt
```

Combine with cut/seq

```
paste <(seq 1 3) <(seq 10 12)
cut -d: -f1 /etc/passwd | head | paste - - -      # 3 columns
```

Join data columns from commands

```
paste <(nproc; echo cores) <(free -h | awk '/Mem:/{print $2}')
```

NUL-safe

```
paste -z -d ' ' file1 file2
```

Classic: make CSV from columns

```
paste -d, col1.txt col2.txt col3.txt > out.csv
```

Notes / Pitfalls

- Unequal line counts: shorter files yield empty fields for missing lines.
- Default tab delimiter can be hard to see — use `-d`, or `cat -A`.
- `-s` changes semantics dramatically (horizontal merge of each file's own lines).
- Not a full CSV writer (no quoting of embedded commas) — use proper CSV tools when needed.
- Large files stream line-by-line; keep inputs aligned intentionally.

2026-relevant notes

- Still great for quick ops glue between two tool outputs.
- For structured data, prefer `jq -s` / `python` when quoting/escaping matters.
- Pair with `column -t` after paste for display.

Related Commands

- `cut` — extract columns
- `join` — merge sorted files on a key
- `awk` — general field processing
- `column` — pretty alignment
- `pr -m` — merge files in columns (different tool)

Additional Resources

- `man paste`

column

Overview

`column` formats text into aligned columns or a table. Ideal for making command output or colon-separated files readable. Commonly used with `mount`, `fstab`-like data, and custom script output. From `util-linux`.

Syntax

```
column [options] [file...]
```

Common Options

Option	Description
<code>-t, --table</code>	Create a table; determine columns automatically
<code>-s sep, --separator</code>	Input separator characters
<code>-o sep, --output-separator</code>	Output separator (default two spaces)
<code>-N names</code>	Column names header (newer)
<code>-R cols</code>	Right-align columns
<code>-J</code>	JSON output (newer <code>util-linux</code>)
<code>-x</code>	Fill columns before rows
<code>-c width</code>	Output width
<code>-n</code>	Disable “merge empty columns” behavior variants
<code>-l num</code>	Max columns keep (see <code>man</code>)

Examples with Explanations

Table-ize whitespace

```
mount | column -t
df -h | column -t
ps -eo pid,user,comm,%cpu --sort=-%cpu | head | column -t
```

Custom separators

```
column -t -s: /etc/passwd | less
echo -e 'a|b|c\n1|2|3' | column -t -s'|'
```

Headers (newer)

```
printf 'user:uid:shell\nalice:1000:/bin/bash\n' | column -t -s: -N USER,UID,SHELL
```

Right-align numbers

```
printf 'name val\na 10\nbb 3\n' | column -t -R 2
```

JSON (when available)

```
mount | column -t -J | jq .
```

Pretty fstab view

```
grep -vE '^\s*(#|$)' /etc/fstab | column -t
```

Pipelines

```
systemctl list-units --type=service --state=running | column -t
```

Notes / Pitfalls

- Input with inconsistent field counts produces jagged tables — clean data first.
- Wide Unicode can misalign in some terminals.
- Default without `-t` wraps differently (classic column mode) — for tables always use `-t`.
- Features like `-N/-J` need sufficiently new util-linux.
- Don't use as a CSV parser for complex quoting — use `csvkit/python` for real CSV.

2026-relevant notes

- Still the fastest path to readable admin tables over SSH.
- JSON mode helps bridge to `jq` tooling when available.
- For TUI tables, tools like `procs`/`eza` already format themselves.

Related Commands

- `awk` / `cut` / `paste` — field extraction/combine
- `rs` — reshape (BSD/some systems)
- `jq -C` / `column` — JSON vs text
- `tabs` / `expand` — tab handling
- `printf` — manual formatting

Additional Resources

- `man column`

jq

Overview

jq is a command-line JSON processor. It slices, filters, maps, and transforms structured data the way `sed/awk` do for text lines. Prefer it for APIs, `package.json`, cloud CLIs, and any tool that emits JSON.

```
sudo apt install jq
```

Syntax

```
jq [options] '<filter>' [file...]  
... | jq [options] '<filter>'
```

Common Options

Option	Description
.	Identity — pretty-print JSON
-r	Raw string output (no JSON quotes)
-c	Compact (one value per line)
-s	Slurp: all inputs into one array
-n	Null input (build JSON from filter only)
-e	Exit non-zero if final result is <code>false</code> or <code>null</code>
--arg name value	Bind string variable <code>\$name</code>
--argjson name json	Bind JSON-valued <code>\$name</code>
-S	Sort object keys
--stream	Stream parse huge inputs (advanced)

Key Use Cases

1. Pretty-print API responses

2. Extract fields for shell scripts
3. Filter and map arrays of objects
4. Build JSON safely from shell variables
5. CI checks that required keys exist

Examples with Explanations

Pretty-print stdin

```
curl -sS https://api.github.com/repos/jqlang/jq | jq .
```

Readable indentation for exploration. `-sS` shows curl errors without progress meter.

Extract a field

```
jq -r .name package.json  
jq -r '.networkSettings.IPAddress // empty' inspect.json
```

`-r` drops quotes so the value is shell-friendly. `// empty` suppresses nulls.

Map a list of objects to TSV

```
jq -r '.items[] | [.id, .name, .status] | @tsv' data.json
```

TSV pipes cleanly into `sort`, `column -t`, or spreadsheets.

Filter array elements

```
jq '.[[] | select(.status=="active" and .role=="admin")' users.json  
jq '.[[] | select(.cpu > 80)] | length' metrics.json
```

`select` keeps matching objects; wrap in `[]` to recollect into an array.

Nested paths and optional keys

```
jq -r '.spec.containers[0].image // "missing"' pod.json
jq 'has("error")' response.json
```

Use explicit paths; `has` is useful for boolean checks with `-e`.

Update / reshape objects

```
jq '.port = 8080 | .debug = false' config.json
jq '{name, version, scripts}' package.json
```

Filters are expressions: assignment returns the modified object; object construction picks fields.

Flatten and walk arrays

```
jq -r '.. | objects | select(has("email")) | .email' big.json
jq -r '.[0] | .tags[]?' items.json
```

? avoids errors on missing paths; `..` is recursive descent (powerful, easy to overuse).

Build JSON from shell (safe)

```
jq -n \
  --arg user "$USER" \
  --arg host "$(hostname -f 2>/dev/null || hostname)" \
  --argjson n "$(nproc)" \
  '{user:$user, host:$host, cpus:$n, ts: now|todateiso8601}'
```

Always prefer `--arg` / `--argjson` over interpolating quotes by hand.

Merge two JSON files

```
jq -s '.[0] * .[1]' base.json overlay.json
```

`-s` slurps files into an array; `*` deep-merges objects (right wins on conflicts).

Exit status for CI

```
jq -e '.version != null' package.json >/dev/null
jq -e '.[ ] | select(.severity=="critical") | halt_error(1)' findings.json
```

`-e` fails on `false`/null final output; `halt_error` aborts mid-stream (jq 1.6+).

Compact logs / NDJSON

```
jq -c '.[ ]' array.json           # one object per line
jq -c -r 'select(.level=="error")' app.ndjson
```

Newline-delimited JSON is ideal for `tail -f` pipelines.

Understanding Output

jq prints the filter result as JSON by default. Multiple results print one after another. Errors go to `stderr`. Parse failures are non-zero. With `-r`, strings lose JSON quoting; numbers/booleans still print in a shell-friendly form.

Notes & Pitfalls

- Filters are **programs** — learn `.`, `[]`, `|`, `select`, `map`, `keys`, `has`, `//`.
- Never build JSON with string concatenation when secrets or user input are involved.
- Huge single JSON documents may need `--stream` or preprocessing.
- YAML is not JSON; use `yq` (or `convert`) for YAML.
- Unicode and locale: raw bytes in invalid UTF-8 can make jq fail.

Related Commands

- `curl` — often the JSON source
- `grep` / `sed` / `awk` — line-oriented text
- `python3 -m json.tool` — pretty-print only
- `yq` — YAML-oriented cousin (if installed)

Additional Resources

- [jq manual](#)
- [jq play](#)

pv

Overview

pv (Pipe Viewer) shows a **progress bar, throughput, and ETA** for data moving through a pipeline. Perfect for long **dd**, compression, or network transfers where tools lack built-in progress.

```
sudo apt install pv
```

Syntax

```
pv [options] [file...]  
... | pv | ...
```

Common Options

Option	Description
-s size	Expected size (better ETA)
-a	Average rate
-r	Current rate
-t	Timer
-b	Bytes transferred
-p	Progress bar
-N name	Label
-c	Cursor progress for multi-pv
-L rate	Rate limit

Examples with Explanations

Copy with progress

```
pv big.iso > /mnt/usb/big.iso
pv big.iso | gzip > big.iso.gz
```

dd-style disk image

```
sudo dd if=/dev/sda bs=4M status=none | pv -s "$(blockdev --getsize64 /dev/sda)" | \
sudo dd of=/dev/sdb bs=4M
```

(Or use `dd status=progress` on modern coreutils.)

Rate-limit a transfer

```
pv -L 10M archive.tar | ssh host 'cat > archive.tar'
```

Multi-stage labeled bars

```
pv -cN in file.tar | gzip | pv -cN gz > file.tar.gz
```

Related Commands

- `dd` — low-level copy
- `rsync --progress` — file sync progress
- `wget/curl` — HTTP progress
- `progress / cv` — attach to running coreutils (if installed)

Additional Resources

- `man pv`

Packages

Intro

Intro

Install and maintain software with distro package managers used on servers. Ubuntu/Debian: **apt**/**dpkg**; Fedora/RHEL: **dnf**/**rpm** (and legacy **yum**); optional Ubuntu **snap** for selected server tools. Desktop app stores (Flatpak, GUI software centers) are out of scope.

Commands in this part

Command	Role
apt	apt is the friendly package manager front-end on Debian and Ubuntu.
dpkg	The dpkg (Debian Package) command is a low-level package manager for Debian-based systems.
dnf	dnf (Dandified YUM) is the primary high-level package manager on Fedora, RHEL 8+, CentOS Stream, Rocky, AlmaLinux,...
rpm	The rpm (RPM Package Manager) command is a low-level package manager for RPM-based Linux distributions.
yum	yum (Yellowdog Updater Modified) is the classic RPM package manager front-end for older RHEL/CentOS/Scientific Linux...
snap	snap installs Snap packages from the Snap Store.

Suggested starting points

1. Debian/Ubuntu day-to-day: **apt** (low-level: **dpkg**).
2. Fedora/RHEL: **dnf** (low-level: **rpm**; legacy: **yum**).
3. Optional on Ubuntu: **snap** only when a server tool requires snapd.

Related parts

- Services and runtime — restart units after library upgrades
- Security — trust and third-party repos
- Files and paths — where binaries land (`which`, `dpkg -L`)

Continue with the individual command pages in this part.

apt

Overview

`apt` is the friendly package manager front-end on Debian and Ubuntu. It installs, updates, searches, and removes `.deb` packages with dependency resolution. Prefer `apt` interactively; keep `apt-get`/`apt-cache` for scripts and older documentation. Fedora/RHEL use `dnf` instead.

Syntax

```
apt [options] command
```

Common Commands

Command	Description
<code>update</code>	Refresh package indexes from mirrors
<code>upgrade</code>	Install newer versions (conservative)
<code>full-upgrade</code>	Upgrade allowing removals to satisfy deps
<code>install PKG...</code>	Install or upgrade packages
<code>reinstall PKG</code>	Reinstall current version
<code>remove / purge</code>	Remove (purge drops conffiles)
<code>autoremove</code>	Drop unused automatic deps
<code>search / show</code>	Discover packages
<code>list --installed / --upgradable</code>	Inventory
<code>policy PKG</code>	Candidate versions / pinning
<code>depends / rdepends</code>	Dependency graph peek
<code>edit-sources</code>	Open sources list
<code>history</code>	Recent transactions (newer apt)
<code>changelog PKG</code>	Show changelog

Key Use Cases

1. Install software on Ubuntu/Debian

2. Apply security updates
3. Inspect why a package is present
4. Clean leftover dependencies after removals

Safety

- Always `update` before `install`/`upgrade` when online.
- Read the transaction summary (packages removed!) before confirming.
- Random PPAs complicate upgrades — pin trust carefully.
- Prefer staged upgrades on production (`apt-get` with maintenance windows).
- Do not kill `dpkg`/`apt` mid-transaction without recovery plan.

Examples with Explanations

Refresh and upgrade

```
sudo apt update
sudo apt upgrade
sudo apt full-upgrade      # may remove packages to satisfy deps
apt list --upgradable
```

Install / remove

```
sudo apt install curl jq htop
sudo apt remove htop
sudo apt purge htop        # also remove /etc config files owned by the pkg
sudo apt autoremove --purge
```

Search and inspect

```
apt search '^nginx'
apt show nginx
apt policy nginx
apt depends nginx
apt rdepends openssl | head
```

Version pinning at install

```
apt policy nginx
sudo apt install nginx=1.24.*
apt-mark hold nginx
apt-mark unhold nginx
apt-mark showhold
```

Fix broken dependencies

```
sudo dpkg --configure -a
sudo apt --fix-broken install
```

Classic recovery after a failed upgrade or full disk mid-install.

Clean caches

```
sudo apt clean          # wipe package caches
sudo apt autoclean      # only obsolete debs
du -sh /var/cache/apt/archives
```

Noninteractive (scripts/CI)

```
export DEBIAN_FRONTEND=noninteractive
sudo apt-get -y update
sudo apt-get -y -o Dpkg::Options::="--force-confdef" \
  -o Dpkg::Options::="--force-confold" upgrade
sudo apt-get -y install package
```

apt-get flags are more stable for automation than interactive apt.

Simulate / dry-run

```
apt-get -s install heavy-metapackage
apt --dry-run full-upgrade
```

Which package owns a file?

```
dpkg -S /usr/bin/curl  
apt-file search bin/curl      # needs apt-file package + update
```

Notes & Pitfalls

- `apt` CLI is **not** a stable machine API — many CI scripts stick to `apt-get`.
- Snaps (`snap`) are a separate delivery channel on some Ubuntu systems; prefer `debs` for core server packages.
- Held packages block upgrades silently until unheld.
- Multi-arch (`:i386` / foreign architectures) appears when enabled for cross-arch deps — uncommon on pure `amd64` servers.
- Offline mirrors: `apt-offline` or local `file:` repositories.

Related Commands

- `dpkg` — low-level `.deb` operations
- `apt-get` / `apt-cache` — classic tooling
- `snap` — optional `snapd` packages on Ubuntu
- `dnf` / `rpm` — Fedora/RHEL family
- `needrestart` — restart services after library upgrades (if installed)

Additional Resources

- `man apt` / `man apt-get`
- Ubuntu Server Guide — package management

dpkg

Overview

The `dpkg` (Debian Package) command is a low-level package manager for Debian-based systems. It directly handles `.deb` package operations without managing dependencies.

Syntax

```
dpkg [options] action
```

Common Actions

Action	Description
<code>-i</code>	Install package
<code>-r</code>	Remove package
<code>-P</code>	Purge package
<code>-l</code>	List packages
<code>-s</code>	Package status
<code>-L</code>	List files
<code>-S</code>	Search file owner
<code>-C</code>	Check database
<code>--configure</code>	Configure package
<code>--unpack</code>	Unpack package
<code>--verify</code>	Verify package
<code>--audit</code>	Audit package

Common Options

Option	Description
<code>--force-all</code>	Force operations
<code>--ignore-depends</code>	Ignore dependencies
<code>--no-act</code>	Simulation mode
<code>--root=dir</code>	Alternative root

Option	Description
<code>--admindir=dir</code>	Alternative admin dir
<code>--log=file</code>	Alternative log file
<code>--status-fd n</code>	Send status to fd n
<code>--print-architecture</code>	Show architecture

Key Use Cases

1. Package installation
2. Package removal
3. Package queries
4. System verification
5. Package information

Examples with Explanations

Example 1: Install Package

```
dpkg -i package.deb
```

Install .deb package

Example 2: Remove Package

```
dpkg -r package_name
```

Remove package

Example 3: List Files

```
dpkg -L package_name
```

List package files

Common Usage Patterns

1. Package status:

```
dpkg -s package_name
```

2. Find owner:

```
dpkg -S /path/to/file
```

3. List installed:

```
dpkg -l | grep '^ii'
```

Security Considerations

1. Package verification
2. Root privileges
3. System integrity
4. Configuration files
5. Dependencies

Related Commands

- `apt` - High-level manager
- `apt-get` - Package management
- `apt-cache` - Package query
- `dselect` - Package selection
- `dpkg-query` - Package database

Additional Resources

- [Dpkg Manual](#)
- [Package Management Guide](#)
- [System Administration](#)

Best Practices

1. Verify packages
2. Backup configuration
3. Check dependencies
4. Document changes

5. Test installation

Package States

1. Not installed
2. Config-files
3. Half-installed
4. Unpacked
5. Half-configured

Troubleshooting

1. Broken packages
2. Dependencies
3. Configuration errors
4. Space issues
5. Database corruption ## Additional Examples

```
dpkg -l | less
dpkg -L bash           # files owned by package
dpkg -S /bin/ls        # which package owns file
sudo dpkg -i package.deb
sudo apt -f install    # fix deps after dpkg -i
```

dnf

Overview

dnf (Dandified YUM) is the primary high-level package manager on **Fedora, RHEL 8+, CentOS Stream, Rocky, AlmaLinux**, and related RPM distributions. This book is Ubuntu-first (**apt**), but operators routinely jump onto RHEL-family hosts — learn **dnf** as the counterpart to **apt**. It installs/removes RPM packages, resolves dependencies, and manages repositories/modules.

On older RHEL/CentOS 7 systems you will still see **yum**; on modern releases **yum** is often a compatibility symlink to **dnf**.

Syntax

```
dnf [options] <command> [<args>...]
```

Common commands

Command	Description
install PKG...	Install packages
remove PKG...	Remove packages
reinstall PKG...	Reinstall
update / upgrade	Apply updates (wording varies by version; both common)
check-update	List available updates without installing
search KEYWORD	Search names/descriptions
info PKG	Package metadata
list	List packages (--installed, --available, ...)
provides / whatprovides	Which package owns a file/feature
repoquery	Advanced query (also dnf repoquery)
repolist	Enabled repositories
makecache	Refresh metadata
clean all	Clear caches
history	Transaction history / undo
group install "Group Name"	Environment/group packages
module	Modular streams (AppStream)
distro-sync	Sync to repo versions (powerful; careful)

Command	Description
<code>autoremove</code>	Remove unneeded deps (when supported)

Common options

Option	Description
<code>-y, --assumeyes</code>	Assume yes
<code>-q</code>	Quiet
<code>--refresh</code>	Force metadata refresh
<code>--enablerepo=ID / --disablerepo=ID</code>	Temporary repo toggles
<code>--setopt=</code>	Override config for this run
<code>--allowerasing</code>	Allow erasing conflicts to satisfy install
<code>--best / --nobest</code>	Strictness of version selection
<code>--security</code>	Prefer security updates where metadata allows
<code>--downloadonly</code>	Fetch without install
<code>--destdir=DIR</code>	With downloadonly, place RPMs here

Safety

- Read the **transaction summary** before accepting (package installs/removals).
- Avoid blind `dnf update -y` on production without a window and rollback plan.
- Third-party repos and modular stream switches can pin you into painful upgrades.
- `dnf history undo` is helpful but not magic after major version jumps or manual `rpm -Uvh` chaos.
- Prefer official AppStream modules over random COPR packages for critical runtimes.

Key Use Cases

1. Install and update software on RHEL-family hosts
2. Map a binary path back to an owning package
3. Apply security updates selectively
4. Review and reverse recent transactions

Examples with Explanations

Example: refresh and update

```
sudo dnf check-update
sudo dnf upgrade
# older muscle memory:
sudo dnf update
```

check-update is a dry look at pending work. Confirm changelogs for kernel and glibc bumps.

Example: install and remove

```
sudo dnf install htop jq curl
sudo dnf remove htop
```

Dependencies are resolved automatically. Removal may leave weak deps depending on config.

Example: search and inspect

```
dnf search nginx
dnf info nginx
dnf list installed 'nginx*'
```

Non-root search/info works for most queries.

Example: which package owns a file

```
dnf provides /usr/bin/ps
dnf provides '*/bin/htop'
```

Invaluable when a command is missing or two packages conflict.

Example: repository visibility

```
dnf repolist
dnf repolist all
sudo dnf config-manager --set-enabled crb # name varies by distro
```

Rocky/Alma often need CodeReady/CRB/PowerTools-style repos for extra build deps.

Example: transaction history

```
dnf history
dnf history info 42
sudo dnf history undo 42
```

Undo a mistaken install when history still maps cleanly to reverse operations.

Example: download only

```
sudo dnf download --resolve nginx
# or
sudo dnf install --downloadonly --destdir=/var/tmp/rpms nginx
```

Useful for air-gapped copies and change tickets that require artifact review.

Example: groups and modules

```
dnf group list
sudo dnf group install "Development Tools"
dnf module list nodejs
sudo dnf module enable nodejs:18
sudo dnf module install nodejs:18/common
```

Modules select streams (versions) on RHEL-family AppStream — read stream notes before switching.

Example: security-focused updates

```
sudo dnf update --security
sudo dnf updateinfo list security
```

Depends on repository updateinfo metadata quality.

Example: clean caches

```
sudo dnf clean all
sudo dnf makecache
```

When metadata looks stale or disk under `/var/cache/dnf` is tight.

Example: scripted noninteractive

```
sudo dnf -y install package-name
```

Still log transactions and pin versions when builds must be reproducible.

Notes & Pitfalls

- This page is for **RPM-family** hosts; on Ubuntu use **apt** / **dpkg**.
- **yum** vs **dnf**: prefer **dnf** on modern systems; check **yum --version** if both exist.
- EPEL is common for extra packages — enable deliberately.
- Kernel updates need reboot; live patching may exist separately (kpatch, vendor tools).
- Mixing **rpm -e --nodeps** with **dnf** creates dependency debt — avoid.
- Proxy/SSL interception breaks metadata fetches; configure **/etc/dnf/dnf.conf** and repo files carefully.

Related Commands

- **rpm** — low-level RPM query/install
- **yum** — legacy / symlink front-end
- **apt** / **dpkg** — Debian/Ubuntu counterparts
- **subscription-manager** — RHEL entitlements
- **needs-restarting** — which services/pids need restart after updates (yum-utils/dnf-utils)

Additional Resources

- **man dnf**
- **man dnf.conf**
- Distribution docs: Fedora DNF guide, RHEL System Administrator's Guide (packaging chapters)

rpm

Overview

The `rpm` (RPM Package Manager) command is a low-level package manager for RPM-based Linux distributions. It handles individual package operations without managing dependencies.

Syntax

```
rpm [options] [package...]
```

Common Options

Option	Description
-i	Install package
-U	Upgrade package
-e	Erase package
-q	Query package
-V	Verify package
-F	Freshen package
--nodeps	Ignore dependencies
--force	Force operation
--test	Test only
--rebuild	Rebuild package
--rebuilddb	Rebuild database
--checksig	Check signature

Query Options

Option	Description
-qa	Query all
-qi	Package info
-ql	List files
-qf	File owner

Option	Description
-qp	Query package file
-qR	Requirements
-qc	Config files
-qd	Documentation

Key Use Cases

1. Package installation
2. Package queries
3. Package verification
4. Database maintenance
5. System verification

Examples with Explanations

Example 1: Install Package

```
rpm -ivh package.rpm
```

Install with verbose and hash progress

Example 2: Query Package

```
rpm -qi package_name
```

Show package information

Example 3: Verify Package

```
rpm -V package_name
```

Verify package files

Common Usage Patterns

1. List installed:

```
rpm -qa
```

2. Find owner:

```
rpm -qf /path/to/file
```

3. Show dependencies:

```
rpm -qR package_name
```

Security Considerations

1. Package verification
2. GPG signatures
3. Root privileges
4. System integrity
5. Dependencies

Related Commands

- yum - High-level manager
- dnf - Next-gen manager
- rpmbuild - Build packages
- rpm2cpio - Convert package
- rpmsign - Sign package

Additional Resources

- [RPM Manual](#)
- [Package Management Guide](#)
- [System Administration](#)

Best Practices

1. Verify packages
2. Check signatures
3. Backup database
4. Document changes

5. Test installation

Package Information

1. Version
2. Release
3. Architecture
4. Dependencies
5. Changelog

Troubleshooting

1. Dependencies
2. Database issues
3. Conflicts
4. Space problems
5. Verification errors

yum

Overview

yum (Yellowdog Updater Modified) is the classic RPM package manager front-end for older RHEL/CentOS/Scientific Linux systems. On **RHEL 8+/Fedora/CentOS Stream**, the command is typically a compatibility wrapper around **dnf**. Prefer **dnf** on modern systems; learn **yum** for legacy fleets and muscle-memory commands that still work.

Syntax

```
yum [options] command
```

Common Commands

Command	Description
install	Install packages
remove / erase	Remove packages
update / upgrade	Update packages
info	Package details
list	List packages
search	Search metadata
provides / whatprovides	Which package owns a capability/file
repolist	Enabled repos
clean all	Clean caches
history	Transaction history
groupinstall	Group install
check-update	Check without installing

Common Options

Option	Description
-y	Assume yes
-q	Quiet

Option	Description
<code>--enablerepo= / --disablerepo=</code>	Toggle repos for one shot
<code>--setopt=</code>	Override config
<code>--downloadonly</code>	Download, don't install
<code>--security</code>	Security updates (plugin/platform dependent)
<code>-C</code>	Run from cache only

Examples with Explanations

Everyday

```
sudo yum check-update
sudo yum update -y
sudo yum install -y curl vim
sudo yum remove -y oldpkg
```

Search and info

```
yum search nginx
yum info nginx
yum list installed 'nginx*'
yum provides /usr/bin/ps
```

Repos

```
yum repolist
yum repolist all
sudo yum-config-manager --enable powertools # platform specific
```

History / undo

```
sudo yum history
sudo yum history info 42
sudo yum history undo 42
```

Clean cache

```
sudo yum clean all
sudo yum makecache
```

Prefer dnf when present

```
command -v dnf && dnf --version
sudo dnf install -y htop      # modern path
# yum may redirect:
yum --version
```

Groups

```
yum group list
sudo yum groupinstall -y "Development Tools"
```

Notes / Pitfalls

- RHEL 8+: `yum` → `dnf` API; some old `yum` plugins differ.
- `-y` in unattended scripts needs good testing (major upgrades).
- Multiple transactions: watch disk space for `/var/cache`.
- Proxy and mirror issues show as metadata download failures.
- EPEL and third-party repos change availability — pin carefully.

2026-relevant notes

- New work: document **dnf** first; mention **yum** as alias/legacy.
- CentOS Linux classic is EOL — care with unmaintained mirrors.
- For containers, use minimal base images and explicit package lists rather than interactive `yum` sessions.

Related Commands

- `dnf` — modern replacement
- `rpm` — low-level RPM queries/installs
- `repoquery` — rich queries (`dnf`/`yum` tools)
- `subscription-manager` — RHEL entitlements
- `apt` — Debian family analog

Additional Resources

- `man yum`, `man dnf`
- RHEL system admin docs for your major version

snap

Overview

`snap` installs **Snap** packages from the Snap Store. On **Ubuntu Server** you mainly meet snaps as optional delivery for server tools (for example some Canonical products, confinement demos, or ops utilities) — not as a desktop app store. Prefer distro packages (`apt` / `dnf`) for core system software; use snaps when a vendor ships only that channel or you want the snap’s isolation/refresh model.

Many minimal server images ship without `snapt`; install it only if you need it.

```
sudo apt update
sudo apt install snapt    # Debian/Ubuntu when required
```

Syntax

```
snap [options] command
```

Common Commands

Command	Description
<code>find query</code>	Search store
<code>info name</code>	Details / channels / confinement
<code>install name</code>	Install
<code>remove name</code>	Remove
<code>list</code>	Installed snaps
<code>refresh</code>	Update
<code>channels name</code>	Track/channel info
<code>services</code>	Services shipped inside snaps
<code>logs name</code>	Snap service logs
<code>connections</code>	Interface plugs/slots (confinement)

Safety

- Snaps **auto-refresh** by default — on production hosts, understand refresh timing or hold refreshes for critical tools.
- Prefer **apt**/**dnf** for libraries and daemons that integrate tightly with the OS.
- Classic confinement is less sandboxed — review **snap info** before install.
- Disk growth under **/var/lib/snapd** from old revisions — prune intentionally.

Key Use Cases

1. Vendor-distributed server tools published only as snaps
2. Optional Ubuntu server components that use snapd
3. Inspecting what snapd already installed on a host

Examples with Explanations

Inventory and search

```
snap version
snap list
snap find lxd
snap info lxd
```

Use **info** to see tracks, confinement, and whether the snap is maintained.

Install / remove (server-oriented)

```
# Example only - choose packages you actually need
sudo snap install hello-world
snap list
sudo snap remove hello-world
```

Prefer well-known server snaps you operate deliberately; avoid random desktop apps on headless hosts.

Channels / tracks

```
snap info kubectl
# sudo snap install kubectl --classic --channel=1.29/stable
```

--classic relaxes confinement (common for developer/ops CLIs). Pin channels on production.

Services and logs

```
snap services
# sudo snap restart <snap>.<service>
snap logs <snap-name>
```

Some snaps bundle systemd-like services managed through snapd.

Hold refreshes (ops)

```
sudo snap refresh --hold
# review: snap refresh --time
# later: sudo snap refresh --unhold
```

Useful in maintenance windows so unattended refreshes do not change binaries mid-incident.

Revisions / disk use

```
snap list --all
du -sh /var/lib/snapd
# remove disabled old revisions when appropriate:
# sudo snap remove <name> --revision=N
```

Notes & Pitfalls

- Headless servers: no GUI plugs; do not install GUI-oriented snaps.
- Mixing snap and apt packages of the same tool can put two binaries on PATH — check `which -a`.
- Network-restricted environments need store/proxy access for install and refresh.
- For Fedora/RHEL, this ecosystem is uncommon; use distro packages or containers instead.

Related Commands

- `apt` / `dpkg` — primary Ubuntu/Debian packaging
- `dnf` / `rpm` — Fedora/RHEL packaging
- `systemctl` / `journalctl` — host services and logs
- `podman` — containers as an alternative distribution model

Additional Resources

- `man snap`
- Ubuntu Server documentation on `snapd` (when using Ubuntu)

Services and Runtime

Intro

Intro

Control the running system under systemd: units, journals, kernel parameters, reboot/shutdown, transient jobs, sessions, modules, and container CLI (**podman**).

Commands in this part

Command	Role
<code>systemctl</code>	<code>systemctl</code> is the primary interface to <code>systemd</code> : start/stop units, enable on boot, inspect status, and list dependencies.
<code>journalctl</code>	<code>journalctl</code> queries the <code>systemd</code> journal — structured logs from the kernel, services, and many applications.
<code>sysctl</code>	<code>sysctl</code> reads and writes kernel parameters exposed under <code>/proc/sys/</code> .
<code>shutdown</code>	<code>shutdown</code> schedules a system power-off, halt, or reboot and notifies logged-in users.
<code>reboot</code>	<code>reboot</code> restarts the system.
<code>podman</code>	<code>podman</code> is a daemonless OCI container engine with a CLI largely compatible with Docker.
<code>systemd-run</code>	<code>systemd-run</code> creates transient units on the fly: one-shot commands, background services, or temporary timers without...
<code>systemd-analyze</code>	<code>systemd-analyze</code> diagnoses boot performance and unit graphs: how long boot took, which units delayed it, whether unit...
<code>loginctl</code>	<code>loginctl</code> manages <code>systemd-logind</code> seats, sessions, and users: who is logged in, idle hints, kill sessions, and...
<code>modprobe</code>	<code>modprobe</code> loads and unloads kernel modules, resolving dependencies via <code>modules.dep</code> — preferred over raw <code>insmod/rmmod</code> ...

Suggested starting points

1. Services: `systemctl status/start/enable`; logs: `journalctl`.
2. Kernel tunables: `sysctl`; modules: `modprobe`.
3. Power: `shutdown/reboot` (prefer coordinated maintenance).
4. Ad-hoc units/timers: `systemd-run`; boot analysis: `systemd-analyze`.
5. User sessions / linger: `loginctl`.
6. Containers: `podman`.

Related parts

- Scheduling — cron vs timers
- Logging — classic log tools alongside the journal
- Processes and jobs — when a unit is really a process tree

Continue with the individual command pages in this part.

systemctl

Overview

systemctl is the primary interface to **systemd**: start/stop units, enable on boot, inspect status, and list dependencies. Ubuntu, Debian, Fedora, and RHEL-family systems all use it for service control.

Syntax

```
systemctl [options] command [unit...]
```

Units use suffixes: `.service`, `.socket`, `.timer`, `.mount`, `.target`, `.path`, ...

Common Commands

Command	Description
<code>status UNIT</code>	State, PIDs, recent journal snippet
<code>start / stop / restart / reload</code>	Lifecycle
<code>enable / disable</code>	Boot linkage
<code>enable --now</code>	Enable and start
<code>is-active / is-enabled / is-failed</code>	Script-friendly checks
<code>mask / unmask</code>	Block start entirely / undo
<code>daemon-reload</code>	Reload unit files after edits
<code>list-units / list-unit-files</code>	Inventory
<code>list-timers / list-sockets</code>	Timers / sockets
<code>cat UNIT</code>	Show unit text (incl. drop-ins)
<code>edit UNIT</code>	Create drop-in override
<code>show UNIT</code>	Low-level properties
<code>reset-failed</code>	Clear failed state
<code>reboot / poweroff / suspend</code>	System actions

Key Use Cases

1. Operate application and infrastructure services

2. Enable services at boot
3. Debug failed units
4. Override vendor unit settings safely with drop-ins
5. Inspect timers as a cron alternative

Safety

- `mask` symlinks a unit to `/dev/null` — easy to forget; prefer `stop + disable` unless you must block activation.
- Do not `restart` remote access (`ssh.service`) on a fleet without console/recovery.
- `daemon-reload` after every unit file change — forgetting it is a common footgun.
- User vs system buses: `systemctl --user` is a separate world.

Examples with Explanations

Status and logs peek

```
systemctl status nginx --no-pager -l
journalctl -u nginx -n 50 --no-pager
```

Start / enable

```
sudo systemctl start nginx
sudo systemctl enable nginx
sudo systemctl enable --now nginx
```

Restart vs reload

```
sudo systemctl reload nginx          # graceful if ExecReload defined
sudo systemctl restart nginx
sudo systemctl try-restart nginx     # only if already active
```


Failed units

```
systemctl --failed
systemctl list-units --state=failed
sudo systemctl reset-failed
```

Script checks

```
systemctl is-active --quiet nginx && echo running
systemctl is-enabled nginx
systemctl is-failed nginx
```

After editing unit files

```
sudo systemctl daemon-reload
sudo systemctl restart myapp.service
```

Drop-in override (preferred over editing vendor files)

```
sudo systemctl edit nginx.service
# creates /etc/systemd/system/nginx.service.d/override.conf
sudo systemctl cat nginx.service
```

Example override snippet:

```
[Service]
Environment=GOMAXPROCS=4
LimitNOFILE=65535
```

Timers (cron alternative)

```
systemctl list-timers --all
systemctl status logrotate.timer
systemctl cat logrotate.timer
```

Dependencies and reverse deps

```
systemctl list-dependencies nginx.service
systemctl list-dependencies --reverse nginx.service
```

User services

```
systemctl --user status podman.socket
loginctl enable-linger "$USER"    # allow user services without login
```

Isolate targets (careful)

```
systemctl get-default
# sudo systemctl isolate multi-user.target    # stops graphical stack, etc.
```

Understanding Output

`status` shows load state (loaded/masked), active state (running/failed/dead), main PID, cgroup path, and a journal snippet. `is-active` / `is-enabled` exit codes are designed for shell scripts (0 = true for the asked condition).

Notes & Pitfalls

- Unit names can be abbreviated when unique (`systemctl status nginx`).
- `restart` is stop+start; open files/sockets may behave differently than `reload`.
- Template units: `systemctl start getty@tty2` instantiates `getty@.service`.
- Masked units look “disabled” but refuse start until `unmask`.
- Containers may run without `systemd` — commands then fail or talk to the host via bind mounts.

Related Commands

- `journalctl` — logs
- `loginctl` / `hostnamectl` / `timedatectl` — other `systemd` tools

- `systemd-analyze` — boot and unit blame
- `systemd-run` — ephemeral units / transient timers
- `service` — legacy wrapper

Additional Resources

- `man systemctl`
- `man systemd.service`
- `man systemd.timer`

journalctl

Overview

journalctl queries the **systemd journal** — structured logs from the kernel, services, and many applications. On systemd systems it replaces a large share of traditional `/var/log/syslog` grepping while adding boot-aware filters and machine-readable output.

Syntax

```
journalctl [options] [matches...]
```

Matches look like `FIELD=value` (e.g. `_SYSTEMD_UNIT=nginx.service`, `_PID=1234`, `PRIORITY=3`).

Common Options

Option	Description
<code>-u UNIT</code>	Messages for unit (repeatable)
<code>-b / -b -1 / -b ID</code>	Current / previous / specific boot
<code>-f</code>	Follow (like <code>tail -f</code>)
<code>-e</code>	Jump to end in pager
<code>-n N</code>	Last N entries
<code>-p PRIORITY / -p from..to</code>	Priority filter (<code>err</code> , <code>warning</code> , 0–7)
<code>-S / -U / --since / --until</code>	Time window
<code>-o short-iso / json / json-pretty / cat</code>	Output format
<code>-k</code>	Kernel messages only
<code>-x</code>	Extra explanations when available
<code>-g REGEX</code>	Message regex filter (newer)
<code>--disk-usage</code>	On-disk journal size
<code>--vacuum-size= / --vacuum-time= / --vacuum-files=</code>	Reclaim space
<code>--no-pager</code>	stdout for scripts
<code>-r</code>	Reverse (newest first)

Key Use Cases

1. Debug a failed service
2. Follow live logs while reproducing a bug
3. Inspect the previous boot after a crash/reboot
4. Export structured logs for jq / SIEM

Examples with Explanations

Unit logs

```
journalctl -u ssh.service -n 50 --no-pager
journalctl -u nginx --since "1 hour ago"
journalctl -u nginx -u php8.3-fpm --since today
```

Follow

```
journalctl -u myapp -f
```

Previous boot / crash triage

```
journalctl --list-boots
journalctl -b -1 -p err..alert --no-pager
journalctl -b -1 -u kdump.service
```

Kernel / dmesg-like

```
journalctl -k -b
journalctl -k --since "10 min ago"
```

Time window

```
journalctl --since "2026-08-10 09:00:00" --until "2026-08-10 10:00:00"
journalctl --since -30m
```

Priority

```
journalctl -p err -b
journalctl -p warning..crit -u myapp
```

JSON for jq

```
journalctl -u nginx -o json -n 100 | jq -r '.MESSAGE'
journalctl -u nginx -o json-pretty -n 5
```

Field matches

```
journalctl _COMM=sudo -n 20
journalctl _UID=1000 --since today | head
journalctl CODE_FILE=/usr/bin/something # when apps set structured fields
```

Disk usage and vacuum

```
journalctl --disk-usage
sudo journalctl --vacuum-size=500M
sudo journalctl --vacuum-time=14d
```

Persistent journals grow until vacuum or system policy limits apply.

Catalog / explain

```
journalctl -x -u NetworkManager -n 20
```

Understanding Output

Default opens a pager. Each line includes timestamp, hostname, syslog identifier, PID, and message. Use `--no-pager` in scripts. Privileges: members of `systemd-journal` or `adm` (distro-dependent) can read more than their own user journals.

Notes & Pitfalls

- Without persistent storage (`/var/log/journal`), logs may live only in volatile `/run/log/journal` and vanish on reboot.
- Vacuum carefully on hosts that need long forensic retention.
- Noisy units: `rate-limit` in the service or drop journal rate limits thoughtfully in `journald.conf`.
- `grep` on pager output is inferior to `-u/-g/field` matches for large journals.

Related Commands

- `systemctl` — service control and status
- `dmesg` — kernel ring buffer
- `logger` — write to journal/syslog
- `jq` — parse JSON output
- `last` / `lastb` — login history (not journal)

Additional Resources

- `man journalctl`
- `man journald.conf`
- `man systemd.journal-fields`

sysctl

Overview

sysctl reads and writes **kernel parameters** exposed under `/proc/sys/`. Changes with `-w` apply immediately to the running kernel; persistence requires files under `/etc/sysctl.conf` or `/etc/sysctl.d/*.conf`.

Syntax

```
sysctl [options] [variable[=value] ...]  
sysctl -p [file]
```

Common Options

Option	Description
<code>-a</code>	Dump all
<code>-w var=val</code>	Write (some builds allow <code>var=val</code> alone)
<code>-p [file]</code>	Load from file (default <code>/etc/sysctl.conf</code>)
<code>-e</code>	Ignore unknown keys
<code>-n</code>	Values only
<code>-N</code>	Names only
<code>-r pattern</code>	Filter names by regex

Key Use Cases

1. Enable IP forwarding for routers/Kubernetes nodes
2. Tune `vm.swappiness`, inotify limits, `file-max`
3. Apply security-related knobs (`rp_filter`, `kptr_restrict`)
4. Inspect live vs on-disk config drift

Examples with Explanations

Read

```
sysctl net.ipv4.ip_forward
sysctl -a | rg 'ip_forward|swappiness|file-max'
sysctl -n vm.swappiness
```

Write (runtime)

```
sudo sysctl -w net.ipv4.ip_forward=1
sudo sysctl -w vm.swappiness=10
sudo sysctl -w fs.inotify.max_user_watches=524288
```

Persist

```
# /etc/sysctl.d/99-local.conf
# net.ipv4.ip_forward = 1
# vm.swappiness = 10
sudo sysctl --system
# or
sudo sysctl -p /etc/sysctl.d/99-local.conf
```

One-liner recipes

```
# Forwarding check on a potential router
sysctl net.ipv4.ip_forward net.ipv6.conf.all.forwarding

# Common developer inotify fix
sudo sysctl -w fs.inotify.max_user_watches=524288

# Show non-default-ish network stack subset
sysctl net.core.somaxconn net.ipv4.tcp_tw_reuse 2>/dev/null
```

Notes & Pitfalls

- Typos with `-w` can break networking—prefer dry-run knowledge and staged files.
- Some keys are **read-only** or namespaced in containers (harder/impossible to change).
- Distribution defaults live in multiple `sysctl.d` snippets—use `sysctl --system` order carefully.
- Don't copy random “performance” `sysctl` pastebins without understanding.

2026-relevant notes

- Kubernetes/kube-proxy and CNI docs still require specific sysctls; prefer documented allowlists.
- Prefer **systemd-sysctl** (`sysctl --system`) over editing only `/etc/sysctl.conf`.
- Cgroup v2 and container runtimes own many resource controls that old sysctl guides misattribute.

Related Commands

- `cat /proc/sys/...` — direct access
- `systemctl` — services may set knobs
- `sysctl --system` — apply all conf
- `prlimit` / `ulimit` — process limits (different layer)

Additional Resources

- `man sysctl`, `man sysctl.d`
- [Kernel Documentation/admin-guide/sysctl/](#)

shutdown

Overview

`shutdown` schedules a system power-off, halt, or reboot and notifies logged-in users. On Ubuntu (systemd), it cooperates with **logind/PID 1** rather than implementing an old SysV path alone. Cousins: `reboot`, `poweroff`, `halt`, and `systemctl reboot|poweroff`. Use clear wall messages on multi-user hosts; cancel with `shutdown -c` when you abort a maintenance window.

Syntax

```
sudo shutdown [options] [time] [wall-message]
sudo reboot [options]
sudo poweroff [options]
sudo halt [options]
```

Time forms

Form	Meaning
now	Immediately
+M	M minutes from now (+15)
hh:mm	Clock time today (or tomorrow if past) on many implementations

Common options (shutdown)

Option	Description
-r	Reboot after shutdown sequence
-h	Halt/power off (historically “halt”; with systemd often powers off)
-H	Halt (may leave power on — hardware dependent)
-P	Power off
-c	Cancel a pending shutdown
-k	Send wall messages only (no real shutdown)

Option	Description
<code>--no-wall</code>	Skip wall broadcast (where supported)
<code>-f / -F</code>	Legacy fsck flags — often ignored under systemd

systemd / loginctl cousins

Command	Effect
<code>systemctl reboot</code>	Reboot
<code>systemctl poweroff</code>	Power off
<code>systemctl halt</code>	Halt
<code>systemctl kexec</code>	kexec reboot when configured
<code>systemctl suspend / hibernate</code>	Sleep states
<code>loginctl reboot / poweroff</code>	Session manager entry points

Safety

- Unsaved work is lost; broadcast intent early (`wall`, chat, tickets).
- Remote reboot without **out-of-band console** (iLO, serial, cloud console) risks lockout if network/boot fails.
- Cloud VMs: provider APIs may force-stop differently from graceful ACPI/systemd shutdown — prefer guest-clean shutdown first.
- Cancelling (`-c`) only works if the schedule is still pending — not after the machine is already going down.
- Avoid `reboot -f /` immediate forced paths unless the system is already wedged.

Key Use Cases

1. Graceful reboot after kernel/package updates
2. Timed maintenance windows with user notification
3. Cancel a mistaken schedule
4. Dry-run messaging with `-k`

Examples with Explanations

Example: power off now

```
sudo shutdown -h now
sudo poweroff
sudo systemctl poweroff
```

Equivalent intents on modern Ubuntu; prefer one style and stay consistent in runbooks.

Example: reboot now

```
sudo shutdown -r now
sudo reboot
sudo systemctl reboot
```

Example: delayed reboot with message

```
sudo shutdown -r +15 "kernel update reboot in 15 minutes - save work"
```

Users get wall notifications; new logins may be blocked as the time approaches (systemd/login semantics).

Example: wall-clock schedule

```
sudo shutdown -r 23:30 "nightly maintenance reboot"
```

Handy for agreed maintenance slots. Confirm timezone with `timedatectl`.

Example: cancel

```
sudo shutdown -c
sudo shutdown -c "reboot cancelled - window postponed"
```

Always cancel explicitly if the change is aborted so users are not left expecting downtime.

Example: message only (drill)

```
sudo shutdown -k +10 "DRILL only - no reboot will occur"
```

Tests communication without risking the host.

Example: systemd reboot with wall message

```
sudo systemctl reboot
# scheduled style often still via shutdown(8):
sudo shutdown -r +5 "applying HA failover test"
```

Example: check last reboots

```
last reboot | head
who -b
journalctl --list-boots | tail
```

Forensics after unexpected restarts.

Example: inhibit shutdown (awareness)

```
systemd-inhibit --list
# apps can delay sleep/shutdown for a reason - check during "why won't it reboot?"
```

Example: forced paths (last resort)

```
sudo systemctl reboot -i          # ignore inhibitors (systemd)
# magic SysRq (if enabled) is an emergency tool - not routine ops
```

Use only when orderly shutdown is impossible and data risk is accepted.

Understanding Output

- `shutdown` typically prints confirmation of the scheduled time.
- Broadcast messages appear on terminals of logged-in users.
- Journal will show shutdown targets activating (`shutdown.target`, `reboot.target`).

Notes & Pitfalls

- On `systemd`, many binaries are related; `reboot` may be a symlink into `systemctl` machinery — behavior is unified.
- Containers: `shutdown` inside a container usually does **not** mean what you want for the host; use the orchestrator.
- Virt guests: clean shutdown lets the hypervisor mark the VM stopped cleanly.

- UPS scripts and cloud metadata “maintenance events” may call these tools for you — know who owns the action.
- `shutdown -h` vs `-P` vs `poweroff` nuances matter on bare metal with odd firmware; test in the lab once.

Related Commands

- `systemctl` — reboot/poweroff targets and inhibitors
- `reboot` / `poweroff` / `halt` — short forms
- `wall` — manual broadcast
- `timedatectl` — confirm clock before `hh:mm` schedules
- `last` / `journalctl` — reboot history
- `loginctl` — session-layer power commands

Additional Resources

- `man shutdown`
- `man systemd-shutdown.service` / `man logind.conf`

reboot

Overview

`reboot` restarts the system. On modern `systemd` systems it is typically a symlink or wrapper around **`systemctl reboot`**. Prefer explicit `systemctl` forms in scripts for clarity and options (wall messages, force flags). Always ensure workloads can stop cleanly.

Syntax

```
reboot [options]
systemctl reboot
systemctl reboot --message="..."
```

Common Options

Option / form	Description
<i>(none)</i>	Normal reboot via <code>init/systemd</code>
<code>--help</code>	Help
<code>systemctl reboot -i</code>	Ignore inhibitors (careful)
<code>systemctl reboot --force</code>	Forceful (see <code>man</code>)
<code>systemctl reboot --firmware-setup</code>	Reboot to firmware setup (UEFI) when supported
<code>shutdown -r now</code>	Equivalent classic form
<code>shutdown -r +15 "msg"</code>	Scheduled reboot

Examples with Explanations

Normal reboot

```
sudo reboot
sudo systemctl reboot
sudo shutdown -r now
```


Scheduled

```
sudo shutdown -r +30 "Reboot for kernel update"
sudo shutdown -c                                # cancel
```

With broadcast message

```
sudo systemctl reboot --message="Kernel update; back in 5 minutes"
```

Inhibitors

```
systemd-inhibit --list
sudo systemctl reboot -i                # ignore inhibitors if stuck
```

Firmware setup (UEFI)

```
sudo systemctl reboot --firmware-setup
```

After kernel install

```
# apply new kernel
sudo reboot
uname -r                                # after boot
```

Containers / hosts

```
# inside a container, reboot may be blocked or reboot the host depending on privileges -
# almost always wrong. Restart the container instead:
podman restart myctr
systemctl restart myapp.service
```

Notes / Pitfalls

- Unsaved work dies; notify users (`wall`, tickets) on multi-user systems.
- Force flags can skip clean unmounts — risk filesystem recovery work.
- Cloud VMs: reboot vs stop/start may change ephemeral networking or public IPs depending on provider.
- Serial/console access ready before remote reboot of network-critical hosts.

- reboot during package transactions can leave broken state — finish `apt`/`dnf` first.

2026-relevant notes

- Live kernel patching reduces some reboot urgency; still needed for many updates.
- Immutable OS / image-based systems reboot into new deployments more often — treat as normal.
- Prefer orchestration drain (k8s) before node reboot.

Related Commands

- `shutdown` / `poweroff` / `halt`
- `systemctl reboot` / `poweroff`
- `timedatectl` / `hostnamectl` — post-boot identity
- `journalctl -b` — logs for current boot
- `who` / `w` — who is logged in before reboot

Additional Resources

- `man reboot`, `man systemctl`, `man shutdown`

podman

Overview

podman is a **daemonless** OCI container engine with a CLI largely compatible with Docker. It excels at **rootless** containers and integrates cleanly with systemd via **podman generate systemd** / Quadlet. Prefer it when you want containers without a long-lived root daemon.

```
sudo apt install podman    # or distro equivalent
sudo dnf install podman
```

Syntax

```
podman [global options] command [options]
```

Common Commands

Command	Description
run	Create + start container
ps / ps -a	List containers
images	List images
pull / push	Registry ops
build	Build from Containerfile/Dockerfile
exec	Exec into running container
stop / start / rm / rmi	Lifecycle
logs / inspect	Debug
pod	Pod grouping (K8s-like)
generate systemd	Unit files
compose	Compose support (podman compose / podman-compose—check install)
login	Registry auth

Key Use Cases

1. Rootless development containers

2. Drop-in `docker` CLI workflows (`alias docker=podman` with caveats)
3. `systemd`-managed long-running containers
4. Building images without `dockerd`

Examples with Explanations

Run

```
podman run -it --rm alpine:3.20 sh
podman run -d --name web -p 8080:80 docker.io/library/nginx:alpine
podman ps
podman logs -f web
```

Build

```
podman build -t myapp:dev .
podman images
```

Rootless tips

```
id
podman info
# subuid/subgid must be configured for the user on many distros
grep ^$USER: /etc/subuid /etc/subgid
```

systemd

```
podman create --name web -p 8080:80 nginx:alpine
podman generate systemd --new --name web > ~/.config/systemd/user/container-web.service
systemctl --user daemon-reload
systemctl --user enable --now container-web.service
```

Quadlet units (`.container` files) are the newer preferred approach on current Fedora/RHEL—check current Podman docs.

One-liner recipes

```
# Cleanup
podman container prune
podman image prune -a    # careful

# Docker socket compatibility is optional; prefer native podman
podman run --rm quay.io/podman/hello
```

Notes & Pitfalls

- **Docker Compose v2** (`docker compose`) is Docker's plugin; Podman users often use `podman compose` or `podman-compose`—behavior is not 100% identical. Verify with current docs.
- Rootless port binding under 1024 needs config/workarounds.
- Volume SELinux labels (`:Z`, `:z`) matter on enforcing systems.
- `alias docker=podman` breaks tools that expect Docker API quirks—test.

2026-relevant notes

- Podman 4/5.x generations emphasize Quadlet and improved compose compatibility—**check your distro's version** rather than assuming flags.
- Kubernetes remains the multi-node orchestrator; Podman is local/engine level.
- Prefer short image names only if `registries.conf` short-name mode is understood—fully qualify `docker.io/library/...` when debugging pulls.

Related Commands

- `docker` — daemon-based engine
- `nerdctl` — containerd CLI
- `buildah` — image build focus
- `skopeo` — copy/inspect images
- `systemctl` — service management

Additional Resources

- `man podman`
- `docs.podman.io` (verify current)

systemd-run

Overview

`systemd-run` creates **transient** units on the fly: one-shot commands, background services, or temporary timers without writing unit files. Ideal for resource-limited jobs, ad-hoc schedules, and testing cgroup settings.

Syntax

```
systemd-run [options] command [args...]  
systemd-run --user [options] command...
```

Common Options

Option	Description
<code>--unit=name</code>	Name the transient unit
<code>--property=...</code> / <code>-p</code>	Set unit properties (<code>CPUQuota=</code> , <code>MemoryMax=</code> , ...)
<code>--timer-property=...</code>	For <code>--on-calendar</code> / timers
<code>--on-calendar=</code>	Schedule like a timer
<code>--on-active=</code>	Run after a delay
<code>-t</code> / <code>--pty</code>	PTY for interactive
<code>-P</code> / <code>--pipe</code>	Connect stdin/stdout
<code>--wait</code>	Wait until finish
<code>--collect</code>	Garbage-collect unit after exit
<code>--uid=</code> / <code>--gid=</code>	Run as user/group
<code>--working-directory=</code>	cwd

Safety

- Transient timers/services disappear after reboot (unless you convert to real units).
- Resource properties can OOM-kill aggressively — test with `--user` when possible.
- Privileged properties need root.

Examples with Explanations

One-shot with wait

```
sudo systemd-run --wait --unit=scratch-job \  
    /usr/local/bin/backup.sh  
systemctl status scratch-job
```

CPU/memory limits

```
sudo systemd-run -p CPUQuota=50% -p MemoryMax=512M --unit=heavy \  
    --wait /usr/local/bin/encode.sh
```

Transient timer

```
sudo systemd-run --on-calendar='*-*- 03:00:00' --unit=nightly-backup \  
    /usr/local/bin/backup.sh  
systemctl list-timers | grep nightly
```

User scope

```
systemd-run --user --on-active=10min --notify-me \  
    notify-send 'Timer fired'
```

Interactive shell in a scope

```
systemd-run --pty --same-dir --wait --collect bash
```

Related Commands

- `systemctl` — manage the resulting unit
- `systemd-analyze` — calendar expression help
- `cron` / `at` — classic scheduling
- `nice` / `ionice` / `systemd-cgtop` — resource control neighbors

Additional Resources

- `man systemd-run`
- `man systemd.timer`
- `man systemd.resource-control`

systemd-analyze

Overview

`systemd-analyze` diagnoses **boot performance** and unit graphs: how long boot took, which units delayed it, whether unit files are valid, and whether calendar/timer expressions parse. First tool when “the box boots slowly”.

Syntax

```
systemd-analyze [options] [subcommand]
```

Common Subcommands

Subcommand	Description
<code>time</code>	Firmware/loader/kernel/userspace timings
<code>blame</code>	Units sorted by time to start
<code>critical-chain</code>	Critical path to default target
<code>plot > boot.svg</code>	SVG timeline
<code>verify UNIT...</code>	Static unit file checks
<code>calendar EXPR</code>	Explain calendar schedule
<code>security UNIT</code>	Sandbox exposure score (newer)
<code>dot</code>	Dependency graph (graphviz)

Examples with Explanations

Boot timings

```
systemd-analyze time
systemd-analyze blame | head
systemd-analyze critical-chain
```

Plot

```
systemd-analyze plot > /tmp/boot.svg  
# open in a browser
```

Verify units after editing

```
systemd-analyze verify /etc/systemd/system/myapp.service
```

Calendar expressions (timers)

```
systemd-analyze calendar 'Mon..Fri *-*-* 09:00:00'  
systemd-analyze calendar 'weekly'
```

Security profile (where available)

```
systemd-analyze security ssh.service | head
```

Notes & Pitfalls

- First boot after kernel update can skew blame times.
- Network mounts and cloud-init often dominate critical-chain.
- Containers without systemd won't produce meaningful host boot data.

Related Commands

- `systemctl`
- `journalctl -b`
- `systemd-run`
- `bootctl` — EFI bootloader (where applicable)

Additional Resources

- `man systemd-analyze`

loginctl

Overview

loginctl manages **systemd-logind** seats, sessions, and users: who is logged in, idle hints, kill sessions, and **lingering** (keep user services running after logout). Essential for systemd user instances and multi-seat systems.

Syntax

```
loginctl [options] command [args...]
```

Common Commands

Command	Description
list-sessions	Active sessions
session-status ID	Detail
user-status USER	User's sessions / linger
list-users	Users with sessions
enable-linger / disable-linger	User services without login
terminate-session / kill-session	End session
lock-session	Lock session (where a lock agent exists)
show-session	Low-level properties

Examples with Explanations

Who is logged in

```
loginctl list-sessions
loginctl list-users
who
```

Session details

```
loginctl session-status  
loginctl show-session "$XDG_SESSION_ID" -p Remote -p Type -p State
```

Enable lingering for user services

```
loginctl enable-linger "$USER"  
# or for another account:  
sudo loginctl enable-linger deploy  
systemctl --user status
```

Required for rootless Podman/user timers to survive logout.

Terminate a stale session

```
loginctl list-sessions  
sudo loginctl terminate-session 42
```

Notes & Pitfalls

- Killing sessions drops that session's processes (including SSH shells attached to it).
- Linger creates `/var/lib/systemd/linger/USER`.
- In containers, `logind` may be absent.

Related Commands

- `systemctl --user`
- `who` / `w` / `last`
- `machinectl`

Additional Resources

- `man loginctl`
- `man logind.conf`

modprobe

Overview

`modprobe` loads and unloads **kernel modules**, resolving dependencies via `modules.dep` — preferred over raw `insmod/rmmod` for daily admin. Pair with `lsmod` to see what is loaded and `dmesg` for driver messages.

Syntax

```
sudo modprobe [options] module [module_options]
sudo modprobe -r module
```

Common Options

Option	Description
<code>-r / --remove</code>	Unload module (+ unused deps)
<code>-v</code>	Verbose
<code>-n</code>	Dry-run
<code>-c</code>	Show config
<code>--show-depends</code>	Print dependency set
<code>-f</code>	Force (dangerous)

Safety

- Removing modules in use can fail or disrupt devices (`rmmod` may force more aggressively — still dangerous).
- Blacklist broken modules via `/etc/modprobe.d/` rather than fighting boot.
- Out-of-tree modules must match the running kernel.

Examples with Explanations

Load / unload

```
lsmod | head
sudo modprobe overlay
sudo modprobe -r overlay
```

Dependencies

```
modprobe --show-depends xfs
```

Module parameters

```
# example only - parameters are module-specific
sudo modprobe nfs nfsvers=4.2
modinfo nfs | less
```

Blacklist (persist)

```
echo 'blacklist pcspkr' | sudo tee /etc/modprobe.d/blacklist-pcspkr.conf
```

List available modules

```
find /lib/modules/"$(uname -r)" -name '*.ko*' | head
```

Related Commands

- `lsmod` / `modinfo`
- `rmmod` / `insmod` — low-level
- `dmesg` — load errors
- `uname -r` — running kernel

Additional Resources

- `man modprobe`
- `modules.dep(5)`

Shell Commands

Intro

Intro

Everyday shell built-ins and small utilities for output, history, name resolution, environment, timing, and the directory stack. Deeper programming belongs in **Scripting** → **Bash** (this book).

Commands in this part

Command	Role
<code>echo</code>	<code>echo</code> writes its arguments to standard output, separated by spaces, usually followed by a newline.
<code>printf</code>	<code>printf</code> formats and prints arguments using a format string, like C's <code>printf</code> .
<code>history</code>	<code>history</code> shows the shell's command history list (<code>bash</code>).
<code>type / command</code>	<code>type</code> explains how the shell will resolve a name — alias, keyword, function, builtin, or external file.
<code>export</code>	<code>export</code> marks shell variables for the environment so child processes inherit them.
<code>time</code>	<code>time</code> runs a command and reports how long it took.
<code>pushd / popd / dirs</code>	<code>pushd</code> , <code>popd</code> , and <code>dirs</code> maintain a directory stack in <code>bash</code> .

Suggested starting points

1. Print: `echo`, `printf`.
2. Recall: `history`.
3. What is this name?: `type / command`.
4. Environment: `export` (see also `env` in System information).
5. Measure: `time`; navigate: `pushd/popd/dirs`.

Related parts

- Help and documentation tools — `help` for builtins

- Text and pipes — consume what you print
- Scripting → Bash (this book) — functions, arrays, strict mode

Continue with the individual command pages in this part.

echo

Overview

`echo` writes its arguments to standard output, separated by spaces, usually followed by a newline. It is the everyday way to print status lines in the shell. Behavior of escapes (`-e`) and options differs across shells and `/bin/echo` — for portable formatting prefer `printf`.

This page stays **operator-light**. Shell programming depth lives in the Shell scripting parts of this book.

Syntax

```
echo [options] [string...]
```

Common Options

Option	Description
<code>-n</code>	No trailing newline
<code>-e</code>	Enable backslash escapes (<code>\n</code> , <code>\t</code> , <code>\\</code> , ...) — bash builtin / some echos
<code>-E</code>	Disable escapes (bash default)

BusyBox and non-bash `echo` may ignore or treat flags differently. When it matters, use `printf`.

Examples with Explanations

Print text

```
echo 'Hello'
echo Hello world
echo "User: $USER"
```

Single quotes preserve literals; double quotes expand variables.

No newline

```
echo -n 'Prompt: '  
# useful before read in interactive snippets
```

Escapes (bash)

```
echo -e 'line1\nline2\tcolumn'
```

Prefer `printf 'line1\nline2\tcolumn\n'` for scripts you share.

Redirect / append

```
echo 'enabled=1' > /tmp/flags.conf  
echo 'note' >> /tmp/flags.conf
```

Pipe into other commands

```
echo 'example.com' | dig +short  
echo "$TOKEN" | wc -c
```

Avoid `echo $TOKEN` unquoted (word-splitting/globbering).

Show a path or command you are about to run

```
echo "Installing into $PREFIX"  
echo "+ apt install -y curl"
```

Common in install scripts for operator visibility.

Notes

- `echo` is often a **shell builtin** (type `echo`); `/bin/echo` may differ.
- Arguments starting with `-` can be misread as options — use `printf '%s\n' "$arg"` for arbitrary data.
- `echo *` expands globs; quote when you mean a literal `*`.
- Don't use `echo` to print binary-safe data; use `printf` or `cat`.

Related Commands

- `printf` — reliable formatting
- `cat` — write multi-line or file contents
- `tee` — print and log
- `yes` — repeat a line

Additional Resources

- `help echo` (bash)
- `man echo`

printf

Overview

`printf` formats and prints arguments using a format string, like C's `printf`. Prefer it over `echo` when you need reliable escapes, fixed columns, no unexpected options parsing, or portable scripts. Available as a bash builtin and `/usr/bin/printf` (usually identical enough for ops use).

Keep complex format logic in **Scripting** → **Bash** in this book; here are the operator essentials.

Syntax

```
printf FORMAT [ARGUMENT...]
```

Common Format Pieces

Sequence	Meaning
<code>%s</code>	String
<code>%d / %i</code>	Decimal integer
<code>%x / %X</code>	Hex integer
<code>%f</code>	Floating point
<code>%b</code>	String with backslash escapes interpreted
<code>%%</code>	Literal %
<code>\n \t \\</code>	Newline, tab, backslash (in FORMAT)

Width/precision: `%-20s`, `%08d`, `%.2f`.

Examples with Explanations

Safe newline print

```
printf '%s\n' "$var"  
printf '%s\n' '---' line1 line2
```

Repeats the format as needed for leftover arguments (POSIX).

No trailing newline

```
printf 'Status: %s' ok
```

Columns

```
printf '%-12s %8s\n' NAME SIZE
printf '%-12s %8s\n' root 120G
printf '%-12s %8s\n' home 800G
```

Handy for quick operator tables in scripts.

Zero-pad numbers

```
printf 'ticket-%04d\n' 7
# ticket-0007
```

Interpret escapes in data

```
printf '%b\n' 'line1\nline2'
```

%b vs putting \n only in FORMAT — know which side holds the escapes.

Hex / bytes (light)

```
printf '%02x\n' 255
printf '\\x%x' 42; echo
```

For full hex dumps use `xxd`.

Build a file

```
printf 'user=%s\nrole=%s\n' "$USER" admin > /tmp/who.ini
```

Arbitrary strings that look like options

```
printf '%s\n' -n  
# correctly prints -n (echo -n would swallow it as a flag)
```

Notes

- Exit status is non-zero on write errors; format errors are shell-dependent.
- Floating formats and locales: decimal comma/point can surprise — set `LC_ALL=C` for machine output.
- Prefer `printf '%s\n' "$x"` over `echo "$x"` in production shell snippets.
- Builtin `printf` vs `/usr/bin/printf`: rare differences; stick to common formats.

Related Commands

- `echo` — simple print
- `sprintf` — not a shell command; use command substitution with `printf`
- `column` — align tables from text
- `xxd` — binary/hex dumps

Additional Resources

- `help printf` (bash)
- `man 1 printf`

history

Overview

`history` shows the shell's command history list (bash). Operators use it to recall prior commands, audit what was run in a session, and re-run lines with event designators (`!!`, `!n`). Configuration lives in `HISTFILE`, `HISTSIZE`, `HISTFILESIZE`, and related bash variables — deep customization belongs in shell-config docs.

Syntax

```
history [n]
history -c
history -d offset
history -a|-n|-r|-w [filename]
```

Without options, lists history with numbers.

Common Options

Option	Description
<code>n</code>	Show last <code>n</code> entries
<code>-c</code>	Clear history list
<code>-d offset</code>	Delete entry at offset
<code>-a</code>	Append new lines to <code>\$HISTFILE</code>
<code>-n</code>	Read unseen lines from <code>\$HISTFILE</code>
<code>-r</code>	Read file and replace current history
<code>-w</code>	Write current history to file
<code>-p</code>	Expand history designators (advanced)
<code>-s</code>	Append args as a single history entry

Examples with Explanations

Show recent commands

```
history | tail -20
history 30
```

Search history (interactive bash)

```
# Ctrl-R reverse incremental search (readline)
history | grep ssh
```

Re-run by number / last command

```
!552      # re-run event 552
!!        # re-run last command
sudo !!   # common: repeat last with sudo
```

Be careful with destructive lines — review before firing !!.

Append now (multi-session)

```
history -a
```

Useful so another terminal can `history -n` and see your latest commands.

Clear session history

```
history -c
history -w # optional: persist the empty/cleared state carefully
```

Clearing does not erase other users' audits (journal, `auth.log`, `bash_history` on disk may still exist until overwritten).

Delete one awkward line

```
history | tail
history -d 1042
```

Still may exist in `$HISTFILE` until rewritten — know privacy limits of shell history.

Where history is stored

```
echo "$HISTFILE"          # typically ~/.bash_history
echo "$HISTSIZE" "$HISTFILESIZE"
```

Notes

- Commands prefixed with space may be omitted if `HISTCONTROL=ignorespace` (common).
- `HISTCONTROL=ignoredups` / `erasedups` change what gets stored.
- History is **not** a security log — rootkits and other shells won't appear; privileged ops need proper auditing.
- `zsh` uses a different history system (`fc`, `HISTFILE`); this page is bash-oriented as Ubuntu default login shell for many admins.
- Avoid putting secrets on the command line; they land in history and process tables.

Related Commands

- `type / command` — resolve what will run
- `alias` — expansions that also show up as typed
- `fc` — fix/re-run history ranges (bash)
- `script` — full terminal typescript recording

Additional Resources

- `help history`
- `man bash` (HISTORY section)

type / command

Overview

`type` explains **how the shell will resolve a name** — alias, keyword, function, builtin, or external file. `command` runs a program **bypassing** functions/aliases of the same name (and can query paths). Together they answer: “Why did *that* run?” when `which` alone misleads (it ignores builtins/aliases depending on setup).

Syntax

```
type [-afptP] name [name...]
command [-pVv] command [arg...]
```

Common Options

type (bash)

Option	Description
-a	All locations / matches
-t	Single word: alias , keyword , function , builtin , file
-p	Path to disk executable if file
-P	Force path search even if non-file
-f	Suppress function lookup

command

Option	Description
-v / -V	Describe command (similar spirit to type)
-p	Use default PATH search
(default)	Execute, skipping shell functions/aliases

Examples with Explanations

What is this name?

```
type ls
type type
type echo cd time
type -a python3
```

Typical: `ls` is aliased to `...`, `echo` is a shell builtin, `python3` is `/usr/bin/python3`.

Machine-readable kind

```
type -t ls
# alias
type -t mkdir
# file
```

Useful in scripts choosing behavior.

Path only

```
type -p sha256sum
# /usr/bin/sha256sum
```

Empty if pure builtin/alias without a file.

Run without alias/function

```
command ls          # unaliased ls
command -v ls
command -p ls
```

Compare to which / command -v

```
which ls
type ls
command -v ls
```

`which` is an external program and can disagree with the shell's true resolution — prefer `type/command -v` inside bash.

Debug “wrong binary”

```
type -a node  
echo "$PATH"
```

Multiple file hits show PATH order; the first is used for external commands.

Notes

- Keywords (`if`, `for`) are not executables — `type` reports `keyword`.
- Hashed paths: bash caches external locations (`hash`); after moves, `hash -r` refreshes.
- `command` is also the safe way to call externals from functions named the same.
- Dash/`sh` may have fewer `type` features — check `help type` in your shell.

Related Commands

- `which` / `whereis` — external path helpers
- `alias` — define aliases
- `hash` — bash command path cache
- `export` / `env` — PATH and environment

Additional Resources

- `help type`, `help command`
- `man bash`

export

Overview

`export` marks shell variables for the **environment** so child processes inherit them. Without `export`, a variable is shell-local. Operators use it for `PATH`, `EDITOR`, proxy variables, language settings, and app config in a session. Permanent setup belongs in shell rc files or systemd units — not ad-hoc forever in one terminal.

Syntax

```
export [-fn] [name[=value] ...]  
export -p
```

Common Options

Option	Description
<code>name=value</code>	Set and export in one step
<code>-p</code>	Print all exported variables (declare style)
<code>-n</code>	Remove export property (variable may remain)
<code>-f</code>	Export functions (bash; use sparingly)

Examples with Explanations

Set for children

```
export EDITOR=vim  
export LANG=en_US.UTF-8  
export PATH="$HOME/.local/bin:$PATH"
```

New processes from this shell see these values.

One-shot without permanent export

```
EDITOR=vim git commit
env NODE_ENV=production node app.js
```

Prefix assignment affects one command; **export** is for the rest of the session (or until changed).

Show exports

```
export -p | head
export -p | grep -E 'PATH|HOME|EDITOR'
env | sort
printenv PATH
```

Unexport / unset

```
export -n MYTEMP          # still set, not in environment
unset MYTEMP              # remove entirely
```

Common operator variables

```
export http_proxy=http://proxy.example:3128
export https_proxy=$http_proxy
export no_proxy=localhost,127.0.0.1,.example.com
export DEBIAN_FRONTEND=noninteractive # apt in scripts (careful)
```

Persist for your user (pointer)

```
# add to ~/.bashrc or ~/.profile as appropriate - then open a new shell
echo 'export PATH="$HOME/bin:$PATH"' >> ~/.bashrc
```

System services should use `unit Environment=` / `EnvironmentFile=`, not your interactive exports.

Notes

- Child shells inherit exports; already-running processes do not update retroactively.
- **export** is a **builtin** — see `type export`.
- Secrets in the environment are visible to other processes with same privileges (`/proc/pid/environ`) — prefer files with mode 600 or secret managers for sensitive material.
- Arrays and complex bash structures don't export portably; stick to strings for environment.

- Typo'd `PATH=` without preserving old value can break the session (`command not found`).

Related Commands

- `env` — run with modified environment / print env
- `printenv` — print variables
- `set` — shell options and variables (broader)
- `declare -x` — bash equivalent flavor
- `systemctl` — service environment

Additional Resources

- `help export`
- `man bash (PARAMETERS / ENVIRONMENT)`

time

Overview

`time` runs a command and reports how long it took. Bash provides a **builtin** `time` (pipeline-aware) and Ubuntu also ships `/usr/bin/time` (GNU `time`) with more format options. Use it for quick performance checks of scripts, builds, and one-liners. For deep profiling use `perf`, language profilers, or application metrics.

Syntax

```
time command [args...]          # usually bash builtin
/usr/bin/time [options] command [args...]
```

Common Options

Bash builtin

Controlled by `TIMEFORMAT` (see `help time` / bash manual). No traditional flags like GNU `time`.

GNU `/usr/bin/time` (package `time`)

Option	Description
<code>-v</code>	Verbose resource stats
<code>-p</code>	Portable output (real/user/sys)
<code>-f FORMAT</code>	Custom format string
<code>-o file</code>	Write timing to file
<code>-a</code>	Append to file

If `type time` shows builtin, call `/usr/bin/time` explicitly for GNU features (`sudo apt install time` if missing).

Examples with Explanations

Basic timing

```
time sleep 2
time tar -czf /tmp/t.tgz /var/log
```

Reports **real** (wall clock), **user** (CPU userspace), **sys** (CPU kernel).

Time a pipeline (bash builtin)

```
time du -sh /var/* 2>/dev/null | sort -h | tail
```

Builtin **time** can attribute the pipeline as a whole depending on shell rules.

GNU verbose

```
/usr/bin/time -v sha256sum large.iso
```

Shows max RSS, page faults, voluntary context switches, exit status, etc.

Portable three lines

```
/usr/bin/time -p make -j"${nproc}"
```

Custom format

```
/usr/bin/time -f 'real %e sec | maxrss %M KB' ./build.sh
```

Compare two approaches

```
time gzip -k bigfile
time zstd -k bigfile
```

Wall clock is what operators feel; user+sys helps spot CPU-bound vs I/O-bound work.

Bash TIMEFORMAT

```
TIMEFORMAT='real %3R s; user %3U s; sys %3S s'
time ls -R /usr/share/doc >/dev/null
```

Notes

- **real** can exceed **user+sys** when waiting on disk/network/CPU contention.
- **user+sys** can exceed **real** on multi-threaded commands (CPU time summed across cores).
- Output of **time** often goes to **stderr**, separate from the command's stdout.
- Alias/functions: **time** times the resolved command; use **command time** / full path if shadowed.
- Micro-benchmarks need warm caches and repeats; single runs mislead.

Related Commands

- **perf stat** — hardware performance counters
- **strace -c** — syscall time summary
- **hyperfine** — benchmarking tool (extra install)
- **date +%s.%N** — manual timestamps around blocks
- **timeout** — limit runtime, not measure it

Additional Resources

- **help time**
- **man time** (GNU)
- **man bash** (TIMEFORMAT)

pushd / popd / dirs

Overview

`pushd`, `popd`, and `dirs` maintain a **directory stack** in bash. `pushd` changes directory and pushes the previous location; `popd` returns; `dirs` lists the stack. Handy when hopping between build trees, config dirs, and logs without losing your place. For a single previous directory, `cd -` is enough.

Light operator page — stack gymnastics in large scripts belong in **Scripting** → **Bash** in this book.

Syntax

```
pushd [dir]
pushd +N | -N
popd
popd +N | -N
dirs [-clpv] [+N | -N]
```

Common Options

dirs

Option	Description
-l	Long paths (no ~ abbreviation)
-p	One entry per line
-v	Numbered list
-c	Clear stack (bash)

pushd / popd

Form	Description
pushd dir	Push current, cd to dir
pushd	Swap top two entries
pushd +N / -N	Rotate stack

Form	Description
popd	Pop top and cd there
popd +N	Delete Nth entry without rotating to it (see help)

set -o / shopt can affect cd behavior (cdspell, autocd); stack commands still need valid paths.

Examples with Explanations

Jump and return

```
pwd
pushd /var/log
# ... inspect logs ...
popd
pwd
```

Nested hops

```
pushd /etc/nginx
pushd /etc/ssl/certs
dirs -v
popd
popd
```

List the stack

```
dirs
dirs -v
dirs -l -p
```

Swap two project roots

```
cd ~/src/app
pushd ~/src/infra
pushd           # swap top two
```


Numbered rotation

```
dirs -v  
pushd +2          # bring entry 2 to top and cd
```

Clear

```
dirs -c
```

Notes

- Stack is **per shell session** — not shared across terminals.
- Failed `pushd` to a bad path usually leaves the stack unchanged (bash) — still verify `pwd`.
- Prefer absolute paths in muscle memory when trees are deep and symlinked.
- `CDPATH` can make `pushd some-name` resolve in unexpected directories — know if you set it.
- Fish/zsh have similar but not identical stack commands.

Related Commands

- `cd / pwd` — basic navigation
- `cd -` — previous directory only
- `realpath / readlink -f` — resolve paths
- `tree / ls` — inspect destinations

Additional Resources

- `help pushd`, `help popd`, `help dirs`
- `man bash` (DIRECTORY STACK)

Security

Intro

Intro

SELinux mode and file labels (Fedora/RHEL-family) plus Linux file capabilities. On Ubuntu, AppArmor is more common — still useful to recognize SELinux tools on mixed fleets.

Commands in this part

Command	Role
<code>getenforce</code>	<code>getenforce</code> prints the current SELinux mode: Enforcing, Permissive, or Disabled.
<code>setenforce</code>	<code>setenforce</code> switches SELinux between Enforcing (1) and Permissive (0) until reboot (or until changed again).
<code>restorecon</code>	<code>restorecon</code> restores the default SELinux file contexts for paths based on policy file-context rules.
<code>getcap</code> / <code>setcap</code>	Linux file capabilities grant subsets of root privilege to executables (e.g.

Suggested starting points

1. Mode: `getenforce` / `setenforce` (temporary permissive for triage only).
2. Labels after copy/mv: `restorecon`.
3. Selective privilege on binaries: `getcap` / `setcap`.

Related parts

- Users and groups — `sudo` and accounts
- Files and paths — DAC permissions and ACLs
- Networking — host firewalls

Continue with the individual command pages in this part.

getenforce

Overview

`getenforce` prints the current **SELinux** mode: **Enforcing**, **Permissive**, or **Disabled**. Standard on Fedora/RHEL/CentOS; on Ubuntu SELinux is usually inactive in favor of AppArmor (`aa-status`). Use this before debugging “AVC denied” issues.

```
# RHEL family
sudo dnf install polycoreutils
```

Syntax

```
getenforce
```

Examples with Explanations

Check mode

```
getenforce
# Enforcing | Permissive | Disabled
```

Related status

```
sestatus
getsebool -a | head
```

Notes & Pitfalls

- **Disabled** requires a reboot to re-enable on traditional setups (prefer **permissive** for troubleshooting).
- Containers may show different SELinux visibility than the host.

Related Commands

- `setenforce` — toggle enforcing/permissive
- `sestatus` — richer status
- `restorecon` / `chcon` — file contexts
- `ausearch` / `audit2allow` — AVC triage
- `aa-status` — AppArmor on Ubuntu

Additional Resources

- `man getenforce`
- RHEL SELinux documentation

setenforce

Overview

`setenforce` switches SELinux between **Enforcing** (1) and **Permissive** (0) until reboot (or until changed again). Permissive logs denials without blocking — useful for triage, not a permanent hardening strategy.

Syntax

```
sudo setenforce 1    # Enforcing
sudo setenforce 0    # Permissive
getenforce
```

Safety

- Permissive mode reduces security — use briefly, then return to enforcing.
- Does not enable SELinux if it is fully **Disabled** in the kernel/config; that needs config + reboot.
- Production changes should go through change control; prefer boolean/`semanage` fixes over long-term permissive.

Examples with Explanations

Temporary permissive for debugging

```
getenforce
sudo setenforce 0
# reproduce the app failure; watch audit log
sudo ausearch -m avc -ts recent | tail
sudo setenforce 1
```

Persist mode (config — not setenforce)

```
# /etc/selinux/config  
# SELINUX=enforcing|permissive|disabled
```

Related Commands

- `getenforce` / `sestatus`
- `restorecon` — fix labels
- `semanage` / `setsebool` — lasting policy tweaks
- `audit2allow` — generate local policy modules carefully

Additional Resources

- `man setenforce`

restorecon

Overview

`restorecon` restores the default **SELinux file contexts** for paths based on policy file-context rules. First fix when a moved/copied file is denied because it carries the wrong label (`httpd` cannot read a file labeled `user_home_t`, etc.).

Syntax

```
sudo restorecon [options] path...
```

Common Options

Option	Description
<code>-v</code>	Verbose (show changes)
<code>-R</code>	Recursive
<code>-F</code>	Force reset
<code>-n</code>	Dry-run (no change)
<code>-e dir</code>	Exclude directory

Examples with Explanations

Fix a web tree after rsync/mv

```
sudo restorecon -Rv /var/www/html
ls -Z /var/www/html | head
```

Dry-run

```
sudo restorecon -Rvn /srv/data
```

Single file after manual copy

```
sudo cp ~/index.html /var/www/html/  
sudo restorecon -v /var/www/html/index.html
```

Notes & Pitfalls

- `cp -a` preserves contexts; `mv` within a filesystem keeps labels that may be wrong for the destination type.
- Custom paths may need `semanage fcontext` **before** `restorecon` will apply the right type.
- On Ubuntu/AppArmor systems this tool may be absent or irrelevant.

Related Commands

- `chcon` — temporary/manual context (prefer `restorecon` + `semanage`)
- `ls -Z` / `id -Z` — show contexts
- `setenforce` / `getenforce`
- `semanage fcontext`

Additional Resources

- `man restorecon`
- `man semanage-fcontext`

getcap / setcap

Overview

Linux **file capabilities** grant subsets of root privilege to executables (e.g. bind low ports without full uid 0). **getcap** lists them; **setcap** assigns them. Prefer capabilities over setuid-root when you must elevate a single binary.

```
sudo apt install libcap2-bin
```

Syntax

```
getcap [-r] path...  
sudo setcap cap_spec path  
sudo setcap -r path          # remove
```

Examples with Explanations

Inspect

```
getcap /usr/bin/ping  
getcap -r /usr/bin 2>/dev/null | head
```

Allow binding privileged ports (example)

```
# illustration - understand security impact first  
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/myapp  
getcap /usr/local/bin/myapp
```

Remove capabilities

```
sudo setcap -r /usr/local/bin/myapp
```

Safety

- Capabilities are powerful; `+ep` (effective+permitted) can be equivalent to selective root.
- Package updates may overwrite custom capabilities.
- Prefer systemd `AmbientCapabilities=` / `CapabilityBoundingSet=` for services.

Notes & Pitfalls

- Filesystems must support extended attributes (`xattr`).
- Copy tools may drop capabilities unless preserved.
- Combine with `seccomp`/namespaces for real confinement.

Related Commands

- `capsh` — capability-aware shell helpers
- `systemctl edit` — service capabilities
- `sudo` — broader elevation
- `getfacl` — ACLs (different mechanism)

Additional Resources

- `man getcap` / `man setcap`
- `man capabilities`

Appendices

Intro

Intro

Long-form extras and cheatsheets that do not fit the one-command page model. Use these as quick reference; prefer the command chapters for explained examples.

Commands in this part

Command	Role
Ubuntu Cheatsheet	Quick Ubuntu-oriented CLI reference

Suggested starting points

1. Ubuntu-oriented command cheatsheet for rapid lookup.

Related parts

- All command parts — detailed pages with safety notes
- Book introduction — learning paths

Continue with the individual command pages in this part.

Ubuntu Cheatsheet

Ubuntu Cheatsheet

System

System Information

- `uname -a` - Displays all system information
- `hostnamectl` - Shows current hostname and related details
- `lscpu` - Lists CPU architecture information
- `timedatectl status` - Shows system time

System Monitoring and Management

- `top` - Displays real-time system processes
- `htop` - An interactive process viewer (needs installation)
- `df -h` - Shows disk usage in a human-readable format
- `free -m` - Displays free and used memory in MB
- `kill <process id>` - Terminates a process

Running Commands

- `[command] &` - Runs command in the background
- `jobs` - Displays background commands
- `fg <command number>` - Brings command to the foreground

Service Management

- `sudo systemctl start <service>` - Starts a service
- `sudo systemctl stop <service>` - Stops a service
- `sudo systemctl status <service>` - Checks the status of a service
- `sudo systemctl reload <service>` - Reloads a service's configuration without interrupting its operation
- `journalctl -f` - Follows the journal, showing new log messages in real time
- `journalctl -u <unit_name>` - Displays logs for a specific systemd unit

Cron Jobs and Scheduling

- `crontab -e` - Edits cron jobs for the current user
- `crontab -l` - Lists cron jobs for the current user

Files

File Management

- `ls` - Lists files and directories
- `touch <filename>` - Creates an empty file or updates the last accessed date
- `cp <source> <destination>` - Copies files from source to destination
- `mv <source> <destination>` - Moves files or renames them
- `rm <filename>` - Deletes a file

Directory Navigation

- `pwd` - Displays the current directory path
- `cd <directory>` - Changes the current directory
- `mkdir <dirname>` - Creates a new directory

File Permissions and Ownership

- `chmod [who] [+/-] [permissions] <file>` - Changes file permissions
- `chmod u+x <file>` - Makes a file executable by its owner
- `chown [user]:[group] <file>` - Changes file owner and group

Searching and Finding

- `find [directory] -name <search_pattern>` - Finds files and directories
- `grep <search_pattern> <file>` - Searches for a pattern in files

Archiving and Compression

- `tar -czvf <name.tar.gz> [files]` - Compresses files into a tar.gz archive
- `tar -xvf <name.tar.[gz|bz|xz]> [destination]` - Extracts a compressed tar archive

Text Editing and Processing

- `nano [file]` - Opens a file in the Nano text editor
- `cat <file>` - Displays the contents of a file
- `less <file>` - Displays the paginated content of a file
- `head <file>` - Shows the first few lines of a file
- `tail <file>` - Shows the last few lines of a file
- `awk '{print}' [file]` - Prints every line in a file

Packages

Package Management (APT)

- `sudo apt install <package>` - Installs a package
- `sudo apt install -f --reinstall <package>` - Reinstalls a broken package
- `apt search <package>` - Searches for APT packages
- `apt-cache policy <package>` - Lists available package versions
- `sudo apt update` - Updates package lists
- `sudo apt upgrade` - Upgrades all upgradable packages
- `sudo apt remove <package>` - Removes a package
- `sudo apt purge <package>` - Removes a package and all its configuration files

Package Management (Snap)

- `snap find <package>` - Search for Snap packages
- `sudo snap install <snap_name>` - Installs a Snap package
- `sudo snap remove <snap_name>` - Removes a Snap package
- `sudo snap refresh` - Updates all installed Snap packages
- `snap list` - Lists all installed Snap packages
- `snap info <snap_name>` - Displays information about a Snap package

Users & Groups

User Management

- `w` - Shows which users are logged in
- `sudo adduser <username>` - Creates a new user
- `sudo deluser <username>` - Deletes a user
- `sudo passwd <username>` - Sets or changes the password for a user
- `su <username>` - Switches user
- `sudo passwd -l <username>` - Locks a user account
- `sudo passwd -u <username>` - Unlocks a user password
- `sudo chage <username>` - Sets user password expiration date

Group Management

- `id [username]` - Displays user and group IDs
- `groups [username]` - Shows the groups a user belongs to
- `sudo addgroup <groupname>` - Creates a new group
- `sudo delgroup <groupname>` - Deletes a group

Networking

Network Configuration

- `ip addr show` - Displays network interfaces and IP addresses
- `ip -s link` - Shows network statistics
- `ss -l` - Shows listening sockets
- `ping <host>` - Pings a host and outputs results

Netplan Configuration

- `cat /etc/netplan/*.yaml` - Displays the current Netplan configuration
- `sudo netplan try` - Tests a new configuration for a set period of time
- `sudo netplan apply` - Applies the current Netplan configuration

Firewall Management

- `sudo ufw status` - Displays the status of the firewall
- `sudo ufw enable` - Enables the firewall
- `sudo ufw disable` - Disables the firewall
- `sudo ufw allow <port/service>` - Allows traffic on a specific port or service
- `sudo ufw deny <port/service>` - Denies traffic on a specific port or service
- `sudo ufw delete allow/deny <port/service>` - Deletes an existing rule

SSH and Remote Access

- `ssh <user@host>` - Connects to a remote host via SSH
- `scp <source> <user@host>:<destination>` - Securely copies files between hosts

LXD

LXD is a modern, secure and powerful tool that provides a unified experience for running and managing containers or virtual machines. Visit <https://canonical.com/lxd> for more information.

Basic Setup

- `lxd init` - Initializes LXD before first use

Creating Instances

- `lxc init ubuntu:22.04 <container name>` - Creates a lxc system container (without starting it)
- `lxc launch ubuntu:24.04 <container name>` - Creates and starts a lxc system container
- `lxc launch ubuntu:22.04 <vm name> --vm` - Creates and starts a virtual machine

Managing Instances

- `lxc list` - Lists instances
- `lxc info <instance>` - Shows status information about an instance
- `lxc start <instance>` - Starts an instance
- `lxc stop <instance> [--force]` - Stops an instance
- `lxc delete <instance> [--force|--interactive]` - Deletes an instance

Accessing Instances

- `lxc exec <instance> -- <command>` - Runs a command inside an instance
- `lxc exec <instance> -- bash` - Gets shell access to an instance (if bash is installed)
- `lxc console <instance> [flags]` - Gets console access to an instance
- `lxc file pull <instance>/<instance_filepath> <local_filepath>` - Pulls a file from an instance
- `lxc file push <local_filepath> <instance>/<instance_filepath>` - Pushes a file to an instance

Using Projects

- `lxc project create <project> [--config <option>]` - Creates a project
- `lxc project set <project> <option>` - Configures a project
- `lxc project switch <project>` - Switches to a project

Ubuntu Pro

Ubuntu Pro delivers 10 years of expanded security coverage on top of Ubuntu's Long Term Support (LTS) commitment in addition to management and compliance tooling. Visit <https://ubuntu.com/pro> to register for free on up to five machines.

Activating Ubuntu Pro

- `sudo pro attach <token>` - Attaches your machine to Ubuntu Pro using a specific token

Managing Services

- `sudo pro status` - Displays the status of all Ubuntu Pro services
- `sudo pro enable <service>` - Enables a specific Ubuntu Pro service
- `sudo pro disable <service>` - Disables a specific Ubuntu Pro service

Extended Security Maintenance (ESM)

- `sudo pro enable esm-infra` - Activates Extended Security Maintenance for infrastructure packages
- `sudo pro enable esm-apps` - Activates ESM for applications

Livepatch Service

- `sudo pro enable livepatch` - Enables the Livepatch service for kernel updates without re-booting

FIPS Mode

- `sudo pro enable fips` - Enables FIPS mode for strict cryptographic standards

Updating Configuration

- `sudo pro refresh` - Refreshes the Ubuntu Pro state
- `sudo pro detach` - Detaches the machine from Ubuntu Pro, disabling all services

Editors

Intro

Overview

Overview: Editing on Linux

Linux gives you more than one way to change a file. Some editors exist to survive a broken package manager. Others are full IDEs in the terminal. Multiplexers keep processes alive after SSH drops. Agent runtimes such as **Herd** treat AI coding agents as first-class citizens of that same terminal world.

This chapter orients you so the rest of the book is a map, not a maze.

The three layers

Think in layers rather than “best editor”:

Layer	Job	Examples
Buffer editor	Change text	vi, Vim, Neovim, Helix, nano, Emacs, Zed
Workspace / session	Keep panes and processes alive	tmux, screen, Zellij, Herdr
Companion CLI	Search, navigate, review	rg, fd, fzf, bat, delta, lazygit

You almost always combine one item from each layer.

```
Herdr / tmux / Zellij (session layer)
```

```
nvim / hx          shell / tests / AI
(editor)           agent
```

```
fd | fzf | rg | bat | git delta
```

Why terminal editors still matter

- **SSH and servers:** No display server; your editor must be TUI or remote-aware
- **Rescue and initramfs:** Often only **vi** or **nano** exist
- **Latency:** Modal editors stay efficient on high-latency links
- **Scriptability:** `$EDITOR` hooks for git, kubectl, crontab, sudoedit
- **Agents:** Coding agents live in terminals; workspaces must outlive laptop sleep

Editor families (quick map)

Family	Feel	Best when
vi / Vim / Neovim	Modal, action-oriented	Everywhere, plugins, muscle memory
LazyVim	Neovim distribution	Fast IDE-like Neovim without building from zero
Helix / Kakoune	Selection-first modal	Modern defaults, multi-cursor, less config
nano / Micro	Modeless, discoverable	Quick edits, newcomers
Emacs	Self-contained environment	Deep customization, org, mail, magit
Zed	Fast GUI (native)	Local projects, multiplayer, GPU-accelerated UI

Deep chapters later cover each in turn.

Workspace tools (quick map)

Tool	Strength	Weakness
tmux	Ubiquitous, scriptable, SSH staple	Defaults feel dated; config is a project
GNU screen	Ancient availability	Weaker ergonomics than tmux
Zellij	Friendly defaults, layouts	Less universal on bare servers
Herd	Agent state, detachable agent herds, API	Newer; focused on agent workflows

If you only learn one classic multiplexer, learn **tmux**. If you run multiple AI coding agents, add **Herd**.

A sane default stack (2026)

For day-to-day Linux development:

```
# Editor (pick one)
export EDITOR=nvim          # or hx, micro, code --wait, zed --wait

# Search / browse
# ripgrep, fd, fzf, bat, eza, delta, zoxide

# Session
# tmux for long SSH + builds
```

```
# herdr when juggling Claude Code / Codex / opencode / etc.
```

Example daily flow:

1. **herdr** or **tmux new -s work**
2. Left pane: editor (**nvim**, **hx**, or **Zed** on the host)
3. Right pane: shell, tests, or an agent
4. Detach (**Ctrl-b d** in **tmux**, or **Herdr detach**) without killing jobs

Learning order inside this book

1. **Choosing an editor** — decision criteria
2. **Terminal workspaces** — mental model shared by **tmux**/**Herdr**/**Zellij**
3. **One classic editor** — **vi** or **nano** so you are never stuck
4. **One daily driver** — **Neovim**/**LazyVim**, **Helix**, or **Micro**
5. **tmux** — sessions, windows, panes, copy-mode
6. **Herdr** — agent-aware workspaces
7. **Zed** — if you want a modern GUI on Linux desktops
8. **Modern tools** — upgrade the shell around the editor

Practice goals

After the foundations:

- ☐ Open, edit, save, and exit with **both** **nano** and **vi**
- ☐ Create a **tmux** session, split panes, detach, reattach over SSH
- ☐ Set **\$EDITOR** and use **git commit** so it opens your editor
- ☐ Search a repo with **rg** and open a hit in your editor
- ☐ (Optional) Run two coding agents under **Herdr** and jump to a blocked pane

What “deepen” means here

Each editor chapter aims for **production usefulness**, not marketing blurbs:

- Install on common distros
- Modal/modeless model explained clearly
- Navigation, edit, search, buffers
- Config file locations and a sensible starter config
- Integration with **git**, **LSP** (where relevant), and multiplexers
- Migration notes from adjacent tools

Start with [Choosing an Editor](#) or jump straight to the part that matches your path.

Choosing an Editor

Choosing an Editor

There is no universal best editor. There is a best fit for **constraints** (SSH? rescue? GPU GUI?), **tasks** (one-line config vs multi-package refactor), and **time** you will invest in config and muscle memory.

Decision matrix

Constraint	Prefer
Broken system, single-user mode	vi (often busybox/nvi) or nano
Fresh server, no packages yet	whatever is installed: usually vi/vim/nano
Long SSH session, many panes	Editor + tmux or Herdr
Want IDE features with little config	LazyVim , Helix , Zed , or VSCodium
Love customization / Lisp	Emacs
Hate modes	nano , Micro , Zed/VS Code
Maximum portability of skills	vi family
Multiple AI agents	Herdr + any editor
Low latency typing feel	modal TUI (Vim/Helix) or native GUI (Zed)

Time investment vs power

```
Low setup                                High setup
nano → Micro → Helix → LazyVim → Neovim → Emacs
      ↑                ↑
      Zed              Vim configs
```

- **Low setup:** useful in an afternoon
- **Medium:** productive in a week
- **High:** multi-month craft (full Neovim/Emacs ecosystems)

Helix and LazyVim intentionally sit in the “powerful but opinionated defaults” band. Classic Vim and Emacs reward long investment.

Modal vs modeless

Modeless (nano, Micro, most GUIs): typing always inserts text; commands use chords (**Ctrl**+...) or menus.

Modal (vi, Vim, Neovim, Helix, Kakoune): normal mode is for verbs and motions; insert mode is for typing. Modes feel strange for a day and then become a force multiplier for structured edits.

Selection-first (Helix, Kakoune): select the region, then act. Vim-style is often **action + motion** (**dw** = delete word). Both are modal; the order differs.

GUI vs TUI vs remote

Kind	Examples	Notes
TUI	nvim, hx, nano, emacs -nw	Best over SSH; pair with tmux/Herdr
Native GUI	Zed, gvim, Emacs GUI	Best local UX; use remote mounts/SSHFS carefully
Electron GUI	VS Code / Codium	Huge ecosystem; heavier RAM
Remote bridge	code-server, SSH remote	GUI feel on remote machine

On a headless VPS, prioritize TUI. On a Linux workstation, Zed or a terminal editor inside a nice terminal (Kitty, WezTerm, Ghostty, Alacritty) is common.

Compatibility with the rest of your stack

Ask:

1. Does `$EDITOR` work for git, crontab, kubectl edit?
2. Does it speak **LSP** for your languages?
3. Can it run inside **tmux** without keybinding wars?
4. Can you open it from **fzf** / file managers?
5. For agents: does the tool leave a real PTY (Herdr/tmux care about this)?

Example git wiring:

```
git config --global core.editor "nvim"
# or: "hx"
# or: "zed --wait"
# or: "code --wait"
```

Recommendation cheat sheet

You are...	Start with	Graduate to
New to Linux	nano	Micro or Helix
Sysadmin	vi + nano	Vim + tmux
App developer (terminal)	LazyVim or Helix	custom Neovim
App developer (desktop)	Zed or VSCodium	same + tmux for servers
Polyglot power user	Neovim or Emacs	your own distro
Running AI coding agents	any editor + Herd r	scripted Herdr workspaces

Anti-patterns

- Switching editors every week before learning motions
- Copy-pasting a 2 000-line Neovim config you do not understand
- Using only a GUI and freezing when SSH drops you into `vi`
- Nested multiplexers without a plan (tmux inside Herdr inside tmux)
- Fighting keybindings: leave **prefix** keys alone until you need them

Pick one for thirty days

1. Choose **one** daily editor from this book
2. Choose **one** session tool (tmux for general, Herdr for agents)
3. Install **rg**, **fd**, **fzf**, **bat**
4. Write a half-page cheatsheet of *your* ten commands
5. Only then evaluate a second editor

Next: [Terminal Workspaces](#) for the session layer, then dive into a concrete editor part.

Terminal Workspaces

Terminal Workspaces

Editors change buffers. **Workspaces** change how long processes live, how many shells you juggle, and whether closing a laptop kills a four-hour test or an AI agent.

This chapter is the shared mental model for **tmux**, **screen**, **Zellij**, and **Herdr**. Dedicated chapters go deeper; *modern-tools* covers Zellij in the CLI toolkit chapter.

Problems workspaces solve

Problem	Without workspace	With workspace
SSH disconnect	Shells die	Detach; processes continue
Many tasks	Terminal tabs on the client only	Server-side panes/windows
Pairing / reattach	Lost context	Named sessions
AI agents	One agent per terminal window; easy to lose	Herdr tracks agent state across panes
Layout	Rebuild every login	Scripted layouts

Core concepts (shared vocabulary)

Session

A **session** is a named container for state that outlives a single client connection.

```
# tmux-shaped idea
tmux new -s api
tmux attach -t api
tmux ls
```

Herdr uses a background **server** plus attachable clients; named sessions group workspaces similarly.

Window / tab

A **window** (tmux) or **tab** (Zellij/Herdr) is a full-screen layout unit. You switch windows to change “desktops” inside the session.

Pane

A **pane** is a split region inside a window: editor left, tests right, logs below.

```
editor    shell
          agent
```

Prefix key

Most multiplexers use a **prefix** chord so keybindings do not steal from Vim/Emacs.

Tool	Default prefix (typical)
tmux	Ctrl-b then key
screen	Ctrl-a then key
Zellij	mode-based (Ctrl-p, Ctrl-t, ...)
Herd	Ctrl-b familiar for tmux users; also mouse-first

Rule: if keys “do nothing” in Vim, check you are not inside a multiplexer waiting for a second key after prefix.

Detach / attach

- **Detach:** client disconnects; session keeps running
- **Attach:** reconnect from another terminal or host

```
# tmux
Ctrl-b d          # detach
tmux attach -t api # reattach

# Herd (conceptually)
# detach leaves the server + agents running; run herd again to attach
```

Choosing among multiplexers

```
Need maximum portability on random servers?
tmux (or screen if tmux missing)
```

```
Want nicer defaults and layouts without much config?
Zellij
```

```
Running Claude Code / Codex / opencode / other agent CLIs?  
Herdr (can still live inside or beside tmux)
```

```
Only local GUI terminal tabs?  
Fine for simple work; still learn tmux for remote
```

When not to nest

Avoid:

```
tmux → herdr → tmux → nvim
```

Prefer one outer session manager:

```
herdr  
  nvim  
  agent  
  shell  
  
# or  
  
tmux  
  nvim  
  shell  
optional: herdr only if you need agent UI features
```

Herdr docs note it can run **inside** tmux as the outer terminal environment, but agent detection does not inspect tmux sessions *launched inside* a Herdr pane. Keep the layering intentional.

Editor integration patterns

Pattern A — editor full window

```
Window 1: nvim  
Window 2: shell / tests  
Window 3: docs / top
```

Simple; great with Vim tabs/buffers for files.

Pattern B — editor + shell split

```
Pane left: hx or nvim
Pane right: cargo test / npm run dev
```

Classic pair-programming-with-yourself layout.

Pattern C — agent herd (Herdr)

```
Workspace "api"
  Tab agents: Claude Code | Codex | opencode
  Tab runtime: dev server | logs
```

Herdr surfaces **working** / **blocked** / **idle** so you jump to the agent that needs input.

Pattern D — remote box

```
ssh build-box
tmux attach -t build || tmux new -s build
# or
herdr --remote build-box
```

Copy mode and scrollback

Terminal multiplexers own **scrollback**. Mouse scrolling may enter copy-mode.

tmux essentials:

```
Ctrl-b [          # copy mode
# move with vi keys if mode-keys vi
Space / Enter      # start/end selection (emacs mode defaults vary)
Ctrl-b ]          # paste buffer
```

Configure vi keys:

```
# ~/.tmux.conf
set -g mode-keys vi
set -g status-keys vi
```

Persistence expectations

Event	tmux/screen/Zellij	Herdr
Close terminal emulator	Session survives if server running	Server keeps panes
SSH drop	Session survives	Session survives
Reboot	Dead unless you script restore	Layout restore / agent resume features (see Herdr docs)
kill-server / stop	Everything dies	Everything dies

Do not assume reboot survival without explicit tooling (systemd user services, tmux-resurrect, Herdr session state features).

Minimal practice lab

```
# 1. Session
tmux new -s lab

# 2. Split
Ctrl-b %      # vertical split
Ctrl-b "      # horizontal split
Ctrl-b o      # next pane

# 3. Run something long
ping -c 1000 1.1.1.1

# 4. Detach and kill the terminal window entirely
Ctrl-b d

# 5. Prove it lived
tmux attach -t lab
```

Then repeat a similar flow after reading the **tmux** and **Herdr** parts.

Keybindings to keep conflict-free

If you use Vim/Neovim:

- Prefer tmux prefix **Ctrl-b** (default) or **Ctrl-a** only if you rebind screen habits carefully
- In Neovim, avoid binding the same chords tmux uses without **passthrough**
- For Helix/Zed, conflicts are fewer; still test copy-mode and pane zoom

Next chapters

- **tmux** — the portable standard for SSH and long jobs
- **Herdr** — agent-native workspaces and CLI/API
- **Workflows** — concrete editor + multiplexer recipes
- **Modern tools** — Zellij, prompts, file jumpers, git pagers

The workspace layer is not optional once your work outlives a single shell. Learn detach/attach early; everything else is efficiency.

Vi

Vi: The Original Visual Editor

Vi (Visual Interface) is the original screen-oriented text editor for Unix systems. Created by Bill Joy in 1976, it remains one of the most fundamental and ubiquitous editors in the Unix/Linux world.

Why Learn Vi?

- **Universal availability:** Found on every Linux distribution and Unix system
- **Minimal resource usage:** Works even on minimal Linux installations and embedded systems
- **Foundation knowledge:** Understanding vi helps with vim, neovim, and other vi-like editors
- **Emergency situations:** Often the only editor available in Linux rescue modes and single-user mode
- **System administration:** Essential for Linux server management and configuration
- **POSIX compliance:** Standardized behavior across Linux distributions

Vi Modes

Vi operates in different modes, which is its defining characteristic:

1. Command Mode (Normal Mode)

- Default mode when vi starts
- Used for navigation and text manipulation commands
- Press **Esc** to return to this mode

2. Insert Mode

- Used for typing text
- Enter with **i**, **a**, **o**, or other insert commands
- Exit with **Esc**

3. Command-Line Mode

- Used for file operations and advanced commands
- Enter with **:** from command mode
- Execute with **Enter**

Basic Operations

Starting Vi

```
# Open new file
vi filename.txt

# Open existing file
vi /path/to/file.txt

# Open file at specific line
vi +25 filename.txt

# Open file at first occurrence of pattern
vi +/pattern filename.txt

# Open multiple files
vi file1.txt file2.txt file3.txt
```

Exiting Vi

```
# Save and quit
:wq

# Quit without saving
:q!

# Save file
:w

# Save as new filename
:w newfilename.txt

# Quit (only if no changes)
:q
```

Navigation Commands

Basic Movement

```
# Character movement
h      # Move left
j      # Move down
k      # Move up
l      # Move right

# Word movement
w      # Move to beginning of next word
b      # Move to beginning of previous word
e      # Move to end of current word

# Line movement
O      # Move to beginning of line
^      # Move to first non-blank character
$      # Move to end of line

# Screen movement
H      # Move to top of screen
M      # Move to middle of screen
L      # Move to bottom of screen
```

Advanced Navigation

```
# Page movement
Ctrl+f  # Page forward
Ctrl+b  # Page backward
Ctrl+d  # Half page down
Ctrl+u  # Half page up

# Line jumping
G        # Go to last line
1G       # Go to first line
25G      # Go to line 25
gg       # Go to first line (vim extension)

# Pattern searching
/pattern # Search forward for pattern
?pattern # Search backward for pattern
n        # Next search result
N        # Previous search result
```

```
# Character finding
fx      # Find next 'x' on current line
Fx      # Find previous 'x' on current line
tx      # Move to character before next 'x'
Tx      # Move to character after previous 'x'
;       # Repeat last f, F, t, or T command
,       # Repeat last f, F, t, or T command in reverse
```

Text Editing Commands

Inserting Text

```
i      # Insert before cursor
a      # Insert after cursor
I      # Insert at beginning of line
A      # Insert at end of line
o      # Open new line below current line
O      # Open new line above current line
```

Deleting Text

```
x      # Delete character under cursor
X      # Delete character before cursor
dd      # Delete entire line
D      # Delete from cursor to end of line
dw      # Delete word
db      # Delete word backward
d$      # Delete to end of line
d0      # Delete to beginning of line
```

Changing Text

```
r      # Replace single character
R      # Replace mode (overwrite)
cw      # Change word
cc      # Change entire line
C      # Change from cursor to end of line
s      # Substitute character
S      # Substitute line
```

Copying and Pasting

```
yy  # Yank (copy) entire line
Y   # Yank entire line
yw  # Yank word
y$  # Yank to end of line
p   # Paste after cursor
P   # Paste before cursor
```

Working with Multiple Files

```
# Open next file
:n

# Open previous file
:prev

# List all files
:args

# Edit new file
:e filename.txt

# Switch between files
:b filename
```

Search and Replace

Basic Search

```
# Search forward
/search_term

# Search backward
?search_term

# Case-sensitive search
/\Csearch_term

# Whole word search
/\
```

Search and Replace

```
# Replace first occurrence on current line
:s/old/new/

# Replace all occurrences on current line
:s/old/new/g

# Replace all occurrences in file
:%s/old/new/g

# Replace with confirmation
:%s/old/new/gc

# Replace in specific line range
:1,10s/old/new/g
```

Advanced Features

Marks and Bookmarks

```
# Set mark 'a' at current position
ma

# Go to mark 'a'
'a

# Go to exact position of mark 'a'
`a

# List all marks
:marks
```

Registers

```
# Yank to register 'a'
"ayy

# Paste from register 'a'
"ap

# View all registers
:reg
```

Macros

```
# Start recording macro in register 'a'
qa

# Stop recording
q

# Execute macro 'a'
@a

# Execute macro 'a' 10 times
10@a
```

Configuration

Vi Configuration File (.exrc)

```
# ~/.exrc
set number          # Show line numbers
set autoindent      # Auto-indent new lines
set tabstop=4       # Set tab width
set shiftwidth=4    # Set indent width
set showmatch       # Show matching brackets
set ignorecase      # Ignore case in searches
set wrapscan        # Wrap searches around file
```

Environment Variables

```
# Set default editor
export EDITOR=vi
export VISUAL=vi

# Set vi options
export EXINIT='set number autoindent tabstop=4'
```

Real-World Scenarios

Scenario 1: Emergency Linux System Recovery

```
# Linux system in single-user mode, only vi available
# Edit critical configuration file
vi /etc/fstab

# Navigate to problematic line causing boot failure
/UUID=bad-uuid

# Delete the problematic line
dd

# Save and exit
:wq

# Alternative: Fix systemd service file
vi /etc/systemd/system/problematic.service

# Comment out problematic line
I# <Esc>

# Save and exit
:wq

# Reload systemd and reboot
:!systemctl daemon-reload && reboot
```

Scenario 2: Linux Log File Analysis

```
# Open systemd journal or log file
vi /var/log/syslog
# or
vi /var/log/systemd/journal.log

# Go to end of file to see recent entries
G

# Search backward for critical errors
?CRITICAL
?ERROR
?Failed

# Navigate through error occurrences
```



```

N    # Previous error
n    # Next error

# Copy error line for analysis
yy

# Open new file to save errors
:e /tmp/error_analysis.txt

# Paste error
p

# Add timestamp and system info
O
# Type analysis header
System: $(hostname) - $(date)
Error Analysis:
<Esc>

# Save and return to log
:w
:b syslog

```

Scenario 3: Linux System Configuration

```

# Open multiple Linux configuration files
vi /etc/hosts /etc/resolv.conf /etc/hostname

# Edit first file (/etc/hosts)
# Add new host entry for local network
o
192.168.1.100    newserver.local
192.168.1.101    database.local
192.168.1.102    webserver.local

# Save and move to next file
:w
:n

# Edit resolv.conf for DNS configuration
# Change nameserver to Google DNS
/nameserver
cw
nameserver 8.8.8.8<Esc>
o

```

```
nameserver 8.8.4.4

# Save and move to next file
:w
:n

# Edit hostname for system identification
# Replace entire content
:%d
i
production-server

# Save all and exit
:wq

# Verify changes took effect
:!hostnamectl status
```

Vi vs Modern Editors

Advantages of Vi

- **Ubiquity:** Available everywhere
- **Speed:** Fast startup and operation
- **Keyboard-driven:** No mouse required
- **Powerful:** Complex operations with few keystrokes
- **Stable:** Unchanged interface for decades

Limitations

- **Learning curve:** Modal editing is initially confusing
- **Limited features:** No syntax highlighting, plugins, etc.
- **No GUI:** Text-only interface
- **Basic functionality:** Lacks modern IDE features

Tips and Best Practices

Essential Tips

1. **Master the modes:** Understanding when you're in which mode is crucial
2. **Use Esc liberally:** When in doubt, press Esc to return to command mode
3. **Learn movement commands:** Efficient navigation is key to vi mastery
4. **Practice regularly:** Vi skills deteriorate without use
5. **Use :help:** Built-in help system is comprehensive

Common Mistakes

1. **Typing in command mode:** Results in unintended commands
2. **Forgetting to save:** Always :w before :q
3. **Case sensitivity:** Commands are case-sensitive
4. **Not using counts:** Numbers multiply commands (5dd deletes 5 lines)

Productivity Shortcuts

```
# Repeat last command
.  
  
# Undo last change  
u  
  
# Join lines  
J  
  
# Change case  
~  
  
# Increment number  
Ctrl+a  
  
# Decrement number  
Ctrl+x
```

Troubleshooting

Common Issues

1. **Can't exit vi**

```
# Press Esc first, then  
:q!
```

2. **Accidentally in replace mode**

```
# Press Esc to return to command mode  
Esc
```

3. **Lost changes**

```
# Check for swap file  
:recover
```

4. File is read-only

```
# Force write
:w!
# Or save as new file
:w newfilename
```

Recovery Options

```
# Recover from swap file
vi -r filename.txt

# List available swap files
vi -r

# Start vi in recovery mode
vi -r
```

Vi remains an essential skill for any Linux user or system administrator. While modern editors offer more features, vi's simplicity, universality, and efficiency make it invaluable for quick edits, system recovery, and situations where other editors aren't available. ## Next steps

- [Vim](#) — everyday improved vi
- [tmux](#) — keep vi sessions alive over SSH
- [Choosing an Editor](#) — when to graduate beyond vi

Vim

Vim: Vi Improved

Vim (Vi Improved) is an enhanced version of the vi editor, created by Bram Moolenaar in 1991. It extends vi with numerous features while maintaining backward compatibility, making it one of the most popular and powerful text editors for developers.

Why Choose Vim?

- **Powerful and extensible:** Thousands of plugins and customizations
- **Efficient editing:** Modal editing with powerful commands
- **Native Linux integration:** Seamlessly integrates with Linux workflows
- **Active community:** Large ecosystem of plugins and configurations
- **Programming-focused:** Excellent support for Linux development workflows
- **Highly customizable:** Extensive configuration options
- **System administration:** Perfect for Linux server management and configuration

Installation

Linux Distributions

```
# Ubuntu/Debian
sudo apt update
sudo apt install vim

# CentOS/RHEL/Fedora
sudo yum install vim
# or
sudo dnf install vim

# Arch Linux
sudo pacman -S vim

# Alpine Linux
sudo apk add vim
```

Building from Source

```
# Clone Vim repository
git clone https://github.com/vim/vim.git
cd vim

# Configure build
./configure --with-features=huge \
            --enable-multibyte \
            --enable-python3interp \
            --enable-gui=gtk3 \
            --enable-cscope

# Compile and install
make
sudo make install
```

Vim Modes

Vim extends vi's modal concept with additional modes:

1. Normal Mode

- Default mode for navigation and commands
- Press Esc to enter from any other mode

2. Insert Mode

- For typing text
- Enter with i, a, o, I, A, O

3. Visual Mode

- For selecting text
- v for character-wise selection
- V for line-wise selection
- Ctrl+v for block-wise selection

4. Command-Line Mode

- For ex commands
- Enter with :, /, ?

5. Replace Mode

- For overwriting text
- Enter with R

Enhanced Navigation

Advanced Movement

```
# Word movement (enhanced)
w      # Next word start
W      # Next WORD start (whitespace-separated)
b      # Previous word start
B      # Previous WORD start
e      # End of word
E      # End of WORD
ge     # End of previous word

# Paragraph movement
{      # Previous paragraph
}      # Next paragraph

# Sentence movement
(      # Previous sentence
)      # Next sentence

# Function movement (programming)
[[     # Previous function
]]     # Next function
[{     # Previous unmatched {
]}     # Next unmatched }

# Jump list navigation
Ctrl+o  # Jump to previous location
Ctrl+i  # Jump to next location
```

Window and Tab Management

```
# Window splitting
:split filename    # Horizontal split
:vsplit filename   # Vertical split
Ctrl+w s          # Split current window horizontally
Ctrl+w v          # Split current window vertically
```



```

# Window navigation
Ctrl+w h      # Move to left window
Ctrl+w j      # Move to bottom window
Ctrl+w k      # Move to top window
Ctrl+w l      # Move to right window
Ctrl+w w      # Cycle through windows

# Window resizing
Ctrl+w +      # Increase height
Ctrl+w -      # Decrease height
Ctrl+w >      # Increase width
Ctrl+w <      # Decrease width
Ctrl+w =      # Equalize window sizes

# Tab management
:tabnew filename # Open new tab
:tabclose        # Close current tab
gt               # Next tab
gT               # Previous tab
:tabs            # List all tabs

```

Advanced Text Editing

Visual Mode Operations

```

# Select text and perform operations
v          # Start visual selection
V          # Select entire lines
Ctrl+v     # Block selection

# Operations on selected text
d          # Delete selection
y          # Yank (copy) selection
c          # Change selection
>          # Indent selection
<          # Unindent selection
=          # Auto-indent selection

```

Advanced Copy/Paste

```
# Named registers
"ayy          # Yank line to register 'a'
"ap           # Paste from register 'a'
"Ayy          # Append line to register 'a'

# Special registers
"+y           # Yank to system clipboard
"+p           # Paste from system clipboard
"*y           # Yank to selection clipboard
"*p           # Paste from selection clipboard
""p           # Paste from unnamed register
".p           # Paste last inserted text
```

Text Objects

```
# Inner text objects
iw           # Inner word
is           # Inner sentence
ip           # Inner paragraph
it           # Inner tag
i"           # Inner quotes
i(           # Inner parentheses
i{           # Inner braces
i[           # Inner brackets

# Around text objects
aw           # Around word (includes spaces)
as           # Around sentence
ap           # Around paragraph
at           # Around tag
a"           # Around quotes
a(           # Around parentheses
a{           # Around braces
a[           # Around brackets

# Usage examples
diw          # Delete inner word
ci"          # Change inner quotes
yap          # Yank around paragraph
```

Configuration and Customization

Vimrc Configuration

```
" ~/.vimrc - Basic configuration

" General settings
set nocompatible          " Disable vi compatibility
set number                " Show line numbers
set relativenumber        " Show relative line numbers
set ruler                 " Show cursor position
set showcmd               " Show command in status line
set showmatch             " Show matching brackets
set hlsearch              " Highlight search results
set incsearch             " Incremental search
set ignorecase            " Ignore case in search
set smartcase             " Smart case sensitivity
set autoindent            " Auto-indent new lines
set smartindent           " Smart indentation
set expandtab              " Use spaces instead of tabs
set tabstop=4             " Tab width
set shiftwidth=4          " Indent width
set softtabstop=4         " Soft tab width
set wrap                  " Wrap long lines
set linebreak             " Break lines at word boundaries
set scrolloff=5           " Keep 5 lines visible around cursor
set sidescrolloff=5       " Keep 5 columns visible around cursor
set backspace=indent,eol,start " Allow backspace over everything
set wildmenu              " Enhanced command completion
set wildmode=longest:full,full " Command completion mode
set laststatus=2          " Always show status line
set encoding=utf-8        " Use UTF-8 encoding
set fileencoding=utf-8    " File encoding
set termencoding=utf-8    " Terminal encoding

" Visual settings
syntax enable             " Enable syntax highlighting
set background=dark       " Dark background
colorscheme desert        " Color scheme
set cursorline            " Highlight current line
set colorcolumn=80        " Show column at 80 characters

" Search settings
set path+=**              " Search in subdirectories
set wildignore+=*.o,*.obj,*.pyc,*.class " Ignore compiled files
```

```

" Backup and swap settings
set backup " Enable backups
set backupdir=~/.vim/backup " Backup directory
set directory=~/.vim/swap " Swap file directory
set undofile " Persistent undo
set undodir=~/.vim/undo " Undo directory

" Create directories if they don't exist
if !isdirectory($HOME."/.vim/backup")
    call mkdir($HOME."/.vim/backup", "p")
endif
if !isdirectory($HOME."/.vim/swap")
    call mkdir($HOME."/.vim/swap", "p")
endif
if !isdirectory($HOME."/.vim/undo")
    call mkdir($HOME."/.vim/undo", "p")
endif

" Key mappings
let mapleader = "," " Set leader key
nnoremap <leader>w :w<CR> " Quick save
nnoremap <leader>q :q<CR> " Quick quit
nnoremap <leader>x :x<CR> " Quick save and quit
nnoremap <leader>h :noh<CR> " Clear search highlighting
nnoremap <C-h> <C-w>h " Window navigation
nnoremap <C-j> <C-w>j
nnoremap <C-k> <C-w>k
nnoremap <C-l> <C-w>l

" Function key mappings
nnoremap <F2> :set number!<CR> " Toggle line numbers
nnoremap <F3> :set paste!<CR> " Toggle paste mode
nnoremap <F4> :set wrap!<CR> " Toggle line wrapping
nnoremap <F5> :source ~/.vimrc<CR> " Reload vimrc

" Auto commands
autocmd BufWritePre * :%s/\s\+$//e " Remove trailing whitespace on save
autocmd BufRead,BufNewFile *.py set filetype=python
autocmd BufRead,BufNewFile *.js set filetype=javascript
autocmd BufRead,BufNewFile *.html set filetype=html

```

Advanced Vimrc Features

```
" Advanced ~/.vimrc additions

" Status line customization
set statusline=%f          " File name
set statusline+=%m         " Modified flag
set statusline+=%r         " Read-only flag
set statusline+=%h         " Help flag
set statusline+=%w         " Preview flag
set statusline+=%=         " Right align
set statusline+=%y         " File type
set statusline+=[%{&ff}]   " File format
set statusline+=[%{&fenc}] " File encoding
set statusline+=\ %l/%L   " Line number/total lines
set statusline+=\ %c      " Column number
set statusline+=\ %P      " Percentage through file

" Custom functions
function! ToggleBackground()
    if &background == 'dark'
        set background=light
    else
        set background=dark
    endif
endfunction
nnoremap <F6> :call ToggleBackground()<CR>

" Word count function
function! WordCount()
    let s:old_status = v:statusmsg
    let position = getpos(".")
    exe ":silent normal g<c-g>"
    let stat = v:statusmsg
    let s:word_count = 0
    if stat != '--No lines in buffer--'
        let s:word_count = str2nr(split(v:statusmsg)[11])
        let v:statusmsg = s:old_status
    end
    call setpos('.', position)
    return s:word_count
endfunction

" File type specific settings
autocmd FileType python setlocal expandtab shiftwidth=4 softtabstop=4
autocmd FileType javascript setlocal expandtab shiftwidth=2 softtabstop=2
autocmd FileType html setlocal expandtab shiftwidth=2 softtabstop=2
```

```

autocmd FileType css setlocal expandtab shiftwidth=2 softtabstop=2
autocmd FileType yaml setlocal expandtab shiftwidth=2 softtabstop=2
autocmd FileType markdown setlocal wrap linebreak nolist textwidth=0 wrapmargin=0

" Abbreviations
iabbrev teh the
iabbrev adn and
iabbrev seperate separate
iabbrev recieve receive

```

Plugin Management

Using vim-plug

```

# Install vim-plug
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
    https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim

" Add to ~/.vimrc
call plug#begin('~/.vim/plugged')

" Essential plugins
Plug 'tpope/vim-sensible'           " Sensible defaults
Plug 'tpope/vim-surround'           " Surround text objects
Plug 'tpope/vim-commentary'         " Comment/uncomment
Plug 'tpope/vim-fugitive'           " Git integration
Plug 'scrooloose/nerdtree'           " File explorer
Plug 'ctrlpvim/ctrlp.vim'           " Fuzzy file finder
Plug 'vim-airline/vim-airline'       " Status line
Plug 'vim-airline/vim-airline-themes' " Airline themes
Plug 'airblade/vim-gitgutter'        " Git diff in gutter
Plug 'dense-analysis/ale'            " Linting and fixing
Plug 'ycm-core/YouCompleteMe'        " Code completion
Plug 'preservim/tagbar'              " Tag browser
Plug 'junegunn/fzf', { 'do': { -> fzf#install() } }
Plug 'junegunn/fzf.vim'              " FZF integration

" Language-specific plugins
Plug 'vim-python/python-syntax'      " Python syntax
Plug 'pangloss/vim-javascript'       " JavaScript syntax
Plug 'mxw/vim-jsx'                   " JSX syntax
Plug 'rust-lang/rust.vim'            " Rust support
Plug 'fatih/vim-go'                  " Go support

```

```

" Color schemes
Plug 'morhetz/gruvbox'           " Gruvbox theme
Plug 'dracula/vim', { 'as': 'dracula' } " Dracula theme
Plug 'joshdick/onedark.vim'      " One Dark theme

call plug#end()

" Plugin configurations
let g:airline_theme='gruvbox'
let g:gruvbox_contrast_dark='hard'
colorscheme gruvbox

" NERDTree configuration
nnoremap <F7> :NERDTreeToggle<CR>
let NERDTreeShowHidden=1

" CtrlP configuration
let g:ctrlp_map = '<c-p>'
let g:ctrlp_cmd = 'CtrlP'
let g:ctrlp_working_path_mode = 'ra'

" ALE configuration
let g:ale_linters = {
\   'python': ['flake8', 'pylint'],
\   'javascript': ['eslint'],
\}
let g:ale_fixers = {
\   'python': ['autopep8', 'black'],
\   'javascript': ['prettier'],
\}

```

Installing Plugins

```

" In Vim command mode
:PlugInstall    " Install plugins
:PlugUpdate     " Update plugins
:PlugClean      " Remove unused plugins
:PlugStatus     " Check plugin status

```

Programming with Vim

Code Navigation

```
# Tag navigation (requires ctags)
Ctrl+]      # Jump to tag definition
Ctrl+t      # Jump back from tag
:tag function # Jump to specific tag
:tags       # Show tag stack

# Code folding
zf          # Create fold
zo          # Open fold
zc          # Close fold
za          # Toggle fold
zR          # Open all folds
zM          # Close all folds
```

Code Completion

```
# Built-in completion
Ctrl+n      # Next completion
Ctrl+p      # Previous completion
Ctrl+x Ctrl+f # File name completion
Ctrl+x Ctrl+l # Line completion
Ctrl+x Ctrl+k # Dictionary completion
Ctrl+x Ctrl+o # Omni completion
```

Debugging Integration

```
" GDB integration
packadd termdebug
:Termdebug program_name

" Debugging commands
:Break      " Set breakpoint
:Clear      " Clear breakpoint
:Step       " Step into
:Over       # Step over
:Continue   " Continue execution
```


Real-World Development Scenarios

Scenario 1: Python Development Setup

```
" Python-specific vimrc additions
autocmd FileType python setlocal expandtab shiftwidth=4 softtabstop=4
autocmd FileType python setlocal textwidth=79
autocmd FileType python setlocal colorcolumn=80
autocmd FileType python nnoremap <buffer> <F9> :exec '!python' shellescape(@%, 1)<cr>

" Python debugging
autocmd FileType python nnoremap <buffer> <F10> :exec '!python -m pdb' shellescape(@%, 1)<cr>

" Virtual environment detection
py3 << EOF
import os
import sys
if 'VIRTUAL_ENV' in os.environ:
    project_base_dir = os.environ['VIRTUAL_ENV']
    activate_this = os.path.join(project_base_dir, 'bin/activate_this.py')
    exec(open(activate_this).read(), dict(__file__=activate_this))
EOF
```

Scenario 2: Web Development Workflow

```
" Web development configuration
autocmd FileType html,css,javascript setlocal expandtab shiftwidth=2 softtabstop=2

" Live reload setup
autocmd BufWritePost *.html,*.css,*.js silent !browser-sync reload

" Emmet integration
let g:user_emmet_leader_key=', '

" Auto-close tags
let g:closetag_filenames = '*.html,*.xhtml,*.phtml,*.jsx'

" Prettier formatting
autocmd BufWritePre *.js,*.jsx,*.css,*.html PrettierAsync
```

Scenario 3: Linux System Administration

```
" Linux configuration file editing
autocmd BufRead,BufNewFile /etc/* setlocal nobackup nowritebackup
autocmd BufRead,BufNewFile /etc/systemd/system/* setlocal filetype=systemd
autocmd BufRead,BufNewFile *.conf setlocal syntax=conf
autocmd BufRead,BufNewFile /etc/nginx/* setlocal filetype=nginx
autocmd BufRead,BufNewFile /etc/apache2/* setlocal filetype=apache

" Linux log file viewing
autocmd BufRead,BufNewFile /var/log/* setlocal readonly
autocmd BufRead,BufNewFile /var/log/* normal G
autocmd BufRead,BufNewFile /var/log/syslog setlocal filetype=messages
autocmd BufRead,BufNewFile /var/log/auth.log setlocal filetype=messages

" Sudo save function for system files
command! W w !sudo tee % > /dev/null
command! Wsudo w !sudo tee % > /dev/null

" Quick Linux config editing
nnoremap <leader>ev :vsplit $MYVIMRC<cr>
nnoremap <leader>sv :source $MYVIMRC<cr>
nnoremap <leader>eh :vsplit /etc/hosts<cr>
nnoremap <leader>es :vsplit /etc/ssh/sshd_config<cr>
nnoremap <leader>en :vsplit /etc/nginx/nginx.conf<cr>

" Systemd service file templates
autocmd BufNewFile *.service Or ~/.vim/templates/systemd.service
autocmd BufNewFile *.timer Or ~/.vim/templates/systemd.timer

" Linux-specific commands
command! Systemctl !sudo systemctl
command! Journalctl !journalctl
command! Dmesg !dmesg | tail -20
```

Advanced Vim Techniques

Macros and Automation

```
# Recording macros
qa          # Start recording macro 'a'
# ... perform actions ...
q          # Stop recording
```

```

# Playing macros
@a          # Execute macro 'a'
@@          # Repeat last macro
10@a       # Execute macro 'a' 10 times

# Editing macros
:let @a='...' # Edit macro 'a' directly

```

Regular Expressions

```

# Search patterns
/\v<word>      # Very magic mode - exact word
/\c<Word>      # Case insensitive search
/\C<Word>      # Case sensitive search

# Substitution patterns
:%s/\v(\w+)\s+(\w+)/\2 \1/g    # Swap words
:%s/\v^{80}).*/\1/g           # Truncate lines to 80 chars
:%s/\v\s+$//g                 # Remove trailing whitespace

```

Custom Commands

```

" Custom commands in vimrc
command! -nargs=1 Search execute 'vimgrep /<args>/gj **/*.py' | copen
command! Todo execute 'vimgrep /TODO\|FIXME\|XXX/gj **/*' | copen
command! RemoveTrailingSpaces %s/\s+$//e
command! -range=% FormatJSON <line1>,<line2>!python -m json.tool

" Usage
:Search function_name
:Todo
:RemoveTrailingSpaces
:FormatJSON

```

Performance Optimization

Vim Performance Tips

```

" Performance optimizations
set lazyredraw      " Don't redraw during macros
set ttyfast         " Fast terminal connection
set synmaxcol=200    " Limit syntax highlighting column

```

```

set regengine=1          " Use old regex engine for speed

" Disable unused features
set noshowmatch          " Don't show matching brackets
let loaded_matchparen = 1 " Disable matchparen plugin

```

Large File Handling

```

" Large file handling
autocmd BufReadPre * if getfsize(expand("%")) > 10000000 | syntax off | endif

" Function for large files
function! LargeFile()
    syntax off
    set eventignore+=FileType
    setlocal bufhidden=unload
    setlocal buftype=nowrite
    setlocal undolevels=-1
endfunction
autocmd BufReadPre * if getfsize(expand("%")) > 50000000 | call LargeFile() | endif

```

Troubleshooting Common Issues

Performance Problems

```

# Profile Vim startup
vim --startuptime startup.log

# Check what's slowing down Vim
:profile start profile.log
:profile func *
:profile file *
# ... use Vim normally ...
:profile pause
:noautocmd qall!

```

Plugin Conflicts

```

# Start Vim without plugins
vim -u NONE -N

```

```
# Start with minimal config
vim -u ~/.vimrc.minimal

# Debug plugin loading
vim -V9myVim.log
```

Recovery and Backup

```
# Recover from swap file
vim -r filename.txt

# List swap files
vim -r

# Disable swap files (not recommended)
set noswapfile
```

Vim's power lies in its extensibility and efficiency. While it has a steep learning curve, mastering Vim can significantly improve your editing speed and workflow, especially for programming and system administration tasks.

Neovim

Neovim: The Future of Vim

Neovim is a modern fork of Vim that focuses on extensibility, usability, and performance. Created in 2014, it maintains Vim compatibility while adding powerful new features like built-in LSP support, Lua scripting, and better plugin architecture.

Why Choose Neovim?

- **Modern architecture:** Asynchronous job control and better plugin API
- **Built-in LSP:** Native Language Server Protocol support for Linux development
- **Lua scripting:** More powerful and faster than Vimscript
- **Better defaults:** Sensible configuration out of the box
- **Active development:** Regular updates and new features
- **Tree-sitter:** Advanced syntax highlighting and code parsing
- **Embedded terminal:** Built-in terminal emulator perfect for Linux workflows
- **Linux-first design:** Optimized for Linux development environments

Installation

Package Managers

```
# Ubuntu/Debian (latest stable)
sudo apt update
sudo apt install neovim

# Ubuntu/Debian (latest unstable)
sudo add-apt-repository ppa:neovim-ppa/unstable
sudo apt update
sudo apt install neovim

# CentOS/RHEL/Fedora
sudo dnf install neovim

# Arch Linux
sudo pacman -S neovim

# SUSE/openSUSE
sudo zypper install neovim
```

```
# Gentoo
sudo emerge app-editors/neovim
```

From Source

```
# Install dependencies (Ubuntu/Debian)
sudo apt install ninja-build gettext libtool libtool-bin autoconf automake cmake g++ pkg-con

# Clone and build
git clone https://github.com/neovim/neovim
cd neovim
make CMAKE_BUILD_TYPE=RelWithDebInfo
sudo make install
```

AppImage (Portable)

```
# Download AppImage
curl -LO https://github.com/neovim/neovim/releases/latest/download/nvim.appimage
chmod u+x nvim.appimage
sudo mv nvim.appimage /usr/local/bin/nvim
```

Configuration Structure

Neovim uses a different configuration structure than Vim:

```
# Configuration directory
~/.config/nvim/

# Main configuration file
~/.config/nvim/init.lua    # Lua configuration (recommended)
# or
~/.config/nvim/init.vim    # Vimscript configuration

# Plugin directory
~/.local/share/nvim/site/pack/
```


Basic Lua Configuration

init.lua Structure

```
-- ~/.config/nvim/init.lua

-- Basic settings
vim.opt.number = true
vim.opt.relativenumber = true
vim.opt.expandtab = true
vim.opt.shiftwidth = 4
vim.opt.tabstop = 4
vim.opt.softtabstop = 4
vim.opt.smartindent = true
vim.opt.wrap = false
vim.opt.swapfile = false
vim.opt.backup = false
vim.opt.undodir = os.getenv("HOME") .. "/.vim/undodir"
vim.opt.undofile = true
vim.opt.hlsearch = false
vim.opt.incsearch = true
vim.opt.termguicolors = true
vim.opt.scrolloff = 8
vim.opt.signcolumn = "yes"
vim.opt.isfname:append("@-@")
vim.opt.updatetime = 50
vim.opt.colorcolumn = "80"

-- Leader key
vim.g.mapleader = " "

-- Key mappings
vim.keymap.set("n", "<leader>pv", vim.cmd.Ex)
vim.keymap.set("v", "J", ":m '>+1<CR>gv=gv")
vim.keymap.set("v", "K", ":m '<-2<CR>gv=gv")
vim.keymap.set("n", "J", "mzJ`z")
vim.keymap.set("n", "<C-d>", "<C-d>zz")
vim.keymap.set("n", "<C-u>", "<C-u>zz")
vim.keymap.set("n", "n", "nzzzv")
vim.keymap.set("n", "N", "Nzzzv")

-- System clipboard
vim.keymap.set({"n", "v"}, "<leader>y", [["+y]])
vim.keymap.set("n", "<leader>Y", [["+Y]])
vim.keymap.set({"n", "v"}, "<leader>d", [["_d]])
```

```

-- Replace word under cursor
vim.keymap.set("n", "<leader>s", [[:%s/\<<C-r><C-w>\>/<C-r><C-w>/gI<Left><Left><Left>]])

-- Make file executable
vim.keymap.set("n", "<leader>x", "<cmd>!chmod +x %<CR>", { silent = true })

-- Auto commands
local augroup = vim.api.nvim_create_augroup
local autocmd = vim.api.nvim_create_autocmd

local yank_group = augroup('HighlightYank', {})
autocmd('TextYankPost', {
  group = yank_group,
  pattern = '*',
  callback = function()
    vim.highlight.on_yank({
      higroup = 'IncSearch',
      timeout = 40,
    })
  end,
})

-- Remove trailing whitespace on save
local trim_group = augroup('TrimWhitespace', {})
autocmd('BufWritePre', {
  group = trim_group,
  pattern = '*',
  command = '%s/\s\+$//e',
})

```

Plugin Management with Packer

Installing Packer

```

git clone --depth 1 https://github.com/wbthomason/packer.nvim\
~/.local/share/nvim/site/pack/packer/start/packer.nvim

```

Packer Configuration

```

-- ~/.config/nvim/lua/plugins.lua

-- This file can be loaded by calling `lua require('plugins')` from your init.lua

```

```

-- Only required if you have packer configured as `opt`
vim.cmd [[packadd packer.nvim]]

return require('packer').startup(function(use)
  -- Packer can manage itself
  use 'wbthomason/packer.nvim'

  -- Fuzzy finder
  use {
    'nvim-telescope/telescope.nvim', tag = '0.1.4',
    requires = { {'nvim-lua/plenary.nvim'} }
  }

  -- Color schemes
  use({
    'rose-pine/neovim',
    as = 'rose-pine',
    config = function()
      vim.cmd('colorscheme rose-pine')
    end
  })

  use('nvim-treesitter/nvim-treesitter', {run = ':TSUpdate'})
  use('nvim-treesitter/playground')
  use('theprimeagen/harpoon')
  use('mbbill/undotree')
  use('tpope/vim-fugitive')

  -- LSP Support
  use {
    'VonHeikemen/lsp-zero.nvim',
    branch = 'v3.x',
    requires = {
      --- Uncomment these if you want to manage LSP servers from neovim
      {'williamboman/mason.nvim'},
      {'williamboman/mason-lspconfig.nvim'},

      -- LSP Support
      {'neovim/nvim-lspconfig'},
      -- Autocompletion
      {'hrsh7th/nvim-cmp'},
      {'hrsh7th/cmp-nvim-lsp'},
      {'L3MON4D3/LuaSnip'},
    }
  }

  -- File explorer

```

```

use {
  'nvim-tree/nvim-tree.lua',
  requires = {
    'nvim-tree/nvim-web-devicons', -- optional, for file icons
  },
  tag = 'nightly' -- optional, updated every week
}

-- Status line
use {
  'nvim-lualine/lualine.nvim',
  requires = { 'nvim-tree/nvim-web-devicons', opt = true }
}

-- Git integration
use {
  'lewis6991/gitsigns.nvim',
  config = function()
    require('gitsigns').setup()
  end
}

-- Auto pairs
use {
  "windwp/nvim-autopairs",
  config = function() require("nvim-autopairs").setup {} end
}

-- Comments
use {
  'numToStr/Comment.nvim',
  config = function()
    require('Comment').setup()
  end
}

-- Terminal integration
use {"akinsho/toggleterm.nvim", tag = '*', config = function()
  require("toggleterm").setup()
end}

-- Which key
use {
  "folke/which-key.nvim",
  config = function()
    vim.o.timeout = true
    vim.o.timeoutlen = 300
  end
}

```

```

        require("which-key").setup {}
    end
}

-- Indent guides
use "lukas-reineke/indent-blankline.nvim"

-- Buffer line
use {'akinsho/bufferline.nvim', tag = "*", requires = 'nvim-tree/nvim-web-devicons'}

end)

```

Loading Plugins in init.lua

```

-- ~/.config/nvim/init.lua
require('plugins')

-- Plugin configurations
require('telescope').setup{}
require('nvim-tree').setup{}
require('lualine').setup{}
require('bufferline').setup{}

```

Language Server Protocol (LSP) Setup

LSP Configuration

```

-- ~/.config/nvim/lua/lsp-config.lua

local lsp_zero = require('lsp-zero')

lsp_zero.on_attach(function(client, bufnr)
    local opts = {buffer = bufnr, remap = false}

    vim.keymap.set("n", "gd", function() vim.lsp.buf.definition() end, opts)
    vim.keymap.set("n", "K", function() vim.lsp.buf.hover() end, opts)
    vim.keymap.set("n", "<leader>vws", function() vim.lsp.buf.workspace_symbol() end, opts)
    vim.keymap.set("n", "<leader>vd", function() vim.diagnostic.open_float() end, opts)
    vim.keymap.set("n", "[d", function() vim.diagnostic.goto_next() end, opts)
    vim.keymap.set("n", "]d", function() vim.diagnostic.goto_prev() end, opts)
    vim.keymap.set("n", "<leader>vca", function() vim.lsp.buf.code_action() end, opts)
    vim.keymap.set("n", "<leader>vrr", function() vim.lsp.buf.references() end, opts)
    vim.keymap.set("n", "<leader>vrn", function() vim.lsp.buf.rename() end, opts)

```

```

    vim.keymap.set("i", "<C-h>", function() vim.lsp.buf.signature_help() end, opts)
end)

-- Mason setup for LSP server management
require('mason').setup({})
require('mason-lspconfig').setup({
  ensure_installed = {
    'tsserver',
    'rust_analyzer',
    'gopls',
    'pyright',
    'lua_ls',
    'clangd',
  },
  handlers = {
    lsp_zero.default_setup,
    lua_ls = function()
      local lua_opts = lsp_zero.nvim_lua_ls()
      require('lspconfig').lua_ls.setup(lua_opts)
    end,
  }
})

-- Completion setup
local cmp = require('cmp')
local cmp_select = {behavior = cmp.SelectBehavior.Select}

cmp.setup({
  sources = {
    {name = 'path'},
    {name = 'nvim_lsp'},
    {name = 'nvim_lua'},
    {name = 'luasnip', keyword_length = 2},
    {name = 'buffer', keyword_length = 3},
  },
  formatting = lsp_zero.cmp_format(),
  mapping = cmp.mapping.preset.insert({
    ['<C-p>'] = cmp.mapping.select_prev_item(cmp_select),
    ['<C-n>'] = cmp.mapping.select_next_item(cmp_select),
    ['<C-y>'] = cmp.mapping.confirm({ select = true }),
    ['<C-Space>'] = cmp.mapping.complete(),
  }),
})

```

Installing Language Servers

```
# Using Mason (recommended)
:Mason
# Then install servers interactively

# Or install manually
npm install -g typescript-language-server
pip install python-lsp-server
go install golang.org/x/tools/gopls@latest
```

Tree-sitter Configuration

Tree-sitter Setup

```
-- ~/.config/nvim/lua/treesitter-config.lua

require'nvim-treesitter.configs'.setup {
  -- A list of parser names, or "all"
  ensure_installed = {
    "c", "lua", "vim", "vimdoc", "query", "python", "javascript",
    "typescript", "rust", "go", "html", "css", "json", "yaml", "markdown"
  },

  -- Install parsers synchronously (only applied to `ensure_installed`)
  sync_install = false,

  -- Automatically install missing parsers when entering buffer
  auto_install = true,

  highlight = {
    enable = true,
    -- Setting this to true will run `:h syntax` and tree-sitter at the same time.
    additional_vim_regex_highlighting = false,
  },

  indent = {
    enable = true
  },

  incremental_selection = {
    enable = true,
    keymaps = {
      init_selection = "gnn",
    }
  }
}
```

```

        node_incremental = "grn",
        scope_incremental = "grc",
        node_decremental = "grm",
    },
},

textobjects = {
    select = {
        enable = true,
        lookahead = true,
        keymaps = {
            ["af"] = "@function.outer",
            ["if"] = "@function.inner",
            ["ac"] = "@class.outer",
            ["ic"] = "@class.inner",
        },
    },
},
}

```

Advanced Neovim Features

Telescope Configuration

```

-- ~/.config/nvim/lua/telescope-config.lua

local builtin = require('telescope.builtin')

vim.keymap.set('n', '<leader>pf', builtin.find_files, {})
vim.keymap.set('n', '<C-p>', builtin.git_files, {})
vim.keymap.set('n', '<leader>ps', function()
    builtin.grep_string({ search = vim.fn.input("Grep > ") })
end)
vim.keymap.set('n', '<leader>vh', builtin.help_tags, {})

require('telescope').setup{
    defaults = {
        file_ignore_patterns = {
            "node_modules",
            ".git",
            "dist",
            "build"
        },
    },
    mappings = {
        i = {

```



```

        ["<C-h>"] = "which_key"
    }
}
},
pickers = {
    find_files = {
        theme = "dropdown",
    }
},
extensions = {
    fzf = {
        fuzzy = true,
        override_generic_sorter = true,
        override_file_sorter = true,
        case_mode = "smart_case",
    }
}
}

-- Load extensions
require('telescope').load_extension('fzf')

```

Terminal Integration

```

-- ~/.config/nvim/lua/terminal-config.lua

require("toggleterm").setup{
    size = 20,
    open_mapping = [[<c-\>]],
    hide_numbers = true,
    shade_filetypes = {},
    shade_terminals = true,
    shading_factor = 2,
    start_in_insert = true,
    insert_mappings = true,
    persist_size = true,
    direction = "float",
    close_on_exit = true,
    shell = vim.o.shell,
    float_opts = {
        border = "curved",
        winblend = 0,
        highlights = {
            border = "Normal",
            background = "Normal",

```

```

    },
  },
}

-- Terminal keymaps
function _G.set_terminal_keymaps()
  local opts = {buffer = 0}
  vim.keymap.set('t', '<esc>', [[<C-\><C-n>]], opts)
  vim.keymap.set('t', 'jk', [[<C-\><C-n>]], opts)
  vim.keymap.set('t', '<C-h>', [[<Cmd>wincmd h<CR>]], opts)
  vim.keymap.set('t', '<C-j>', [[<Cmd>wincmd j<CR>]], opts)
  vim.keymap.set('t', '<C-k>', [[<Cmd>wincmd k<CR>]], opts)
  vim.keymap.set('t', '<C-l>', [[<Cmd>wincmd l<CR>]], opts)
end

vim.cmd('autocmd! TermOpen term:/* lua set_terminal_keymaps()')

```

Real-World Development Setups

Python Development

```

-- ~/.config/nvim/lua/python-config.lua

-- Python-specific settings
vim.api.nvim_create_autocmd("FileType", {
  pattern = "python",
  callback = function()
    vim.opt_local.expandtab = true
    vim.opt_local.shiftwidth = 4
    vim.opt_local.tabstop = 4
    vim.opt_local.softtabstop = 4
    vim.opt_local.colorcolumn = "88"
  end,
})

-- Python debugging with DAP
local dap = require('dap')
dap.adapters.python = {
  type = 'executable',
  command = 'python',
  args = { '-m', 'debugpy.adapter' },
}

dap.configurations.python = {

```

```

{
  type = 'python',
  request = 'launch',
  name = "Launch file",
  program = "${file}",
  pythonPath = function()
    return '/usr/bin/python'
  end,
},
}

-- Python testing
vim.keymap.set('n', '<leader>pt', ':!python -m pytest %<CR>')
vim.keymap.set('n', '<leader>pr', ':!python %<CR>')

```

JavaScript/TypeScript Development

```

-- ~/.config/nvim/lua/js-config.lua

-- JavaScript/TypeScript settings
vim.api.nvim_create_autocmd("FileType", {
  pattern = {"javascript", "typescript", "javascriptreact", "typescriptreact"},
  callback = function()
    vim.opt_local.expandtab = true
    vim.opt_local.shiftwidth = 2
    vim.opt_local.tabstop = 2
    vim.opt_local.softtabstop = 2
  end,
})

-- ESLint and Prettier integration
local null_ls = require("null-ls")
null_ls.setup({
  sources = {
    null_ls.builtins.formatting.prettier,
    null_ls.builtins.diagnostics.eslint,
    null_ls.builtins.code_actions.eslint,
  },
})

-- Auto-format on save
vim.api.nvim_create_autocmd("BufWritePre", {
  pattern = {"*.js", "*.jsx", "*.ts", "*.tsx"},
  callback = function()
    vim.lsp.buf.format()
  end,
})

```

```
    end,  
  })
```

Go Development

```
-- ~/.config/nvim/lua/go-config.lua  
  
-- Go settings  
vim.api.nvim_create_autocmd("FileType", {  
  pattern = "go",  
  callback = function()  
    vim.opt_local.expandtab = false  
    vim.opt_local.tabstop = 4  
    vim.opt_local.shiftwidth = 4  
  end,  
})  
  
-- Go-specific keymaps  
vim.api.nvim_create_autocmd("FileType", {  
  pattern = "go",  
  callback = function()  
    local opts = { buffer = true }  
    vim.keymap.set('n', '<leader>gr', ':!go run %<CR>', opts)  
    vim.keymap.set('n', '<leader>gt', ':!go test<CR>', opts)  
    vim.keymap.set('n', '<leader>gb', ':!go build<CR>', opts)  
    vim.keymap.set('n', '<leader>gf', ':!gofmt -w %<CR>', opts)  
  end,  
})
```

Debugging with DAP

DAP Configuration

```
-- ~/.config/nvim/lua/dap-config.lua  
  
local dap = require('dap')  
local dapui = require('dapui')  
  
-- DAP UI setup  
dapui.setup()  
  
-- Auto-open/close DAP UI  
dap.listeners.after.event_initialized["dapui_config"] = function()
```

```

    dapui.open()
end
dap.listeners.before.event_terminated["dapui_config"] = function()
    dapui.close()
end
dap.listeners.before.event_exited["dapui_config"] = function()
    dapui.close()
end

-- Keymaps
vim.keymap.set('n', '<F5>', dap.continue)
vim.keymap.set('n', '<F10>', dap.step_over)
vim.keymap.set('n', '<F11>', dap.step_into)
vim.keymap.set('n', '<F12>', dap.step_out)
vim.keymap.set('n', '<leader>b', dap.toggle_breakpoint)
vim.keymap.set('n', '<leader>B', function()
    dap.set_breakpoint(vim.fn.input('Breakpoint condition: '))
end)
vim.keymap.set('n', '<leader>lp', function()
    dap.set_breakpoint(nil, nil, vim.fn.input('Log point message: '))
end)
vim.keymap.set('n', '<leader>dr', dap.repl.open)
vim.keymap.set('n', '<leader>dl', dap.run_last)

```

Performance Optimization

Startup Optimization

```

-- ~/.config/nvim/lua/performance.lua

-- Disable unused providers
vim.g.loaded_python_provider = 0
vim.g.loaded_ruby_provider = 0
vim.g.loaded_perl_provider = 0
vim.g.loaded_node_provider = 0

-- Lazy load plugins
vim.opt.runtimepath:prepend(vim.fn.stdpath("data") .. "/lazy/lazy.nvim")

-- Optimize for large files
vim.api.nvim_create_autocmd("BufReadPre", {
    callback = function()
        local file_size = vim.fn.getfsize(vim.fn.expand("%"))
        if file_size > 1024 * 1024 then -- 1MB

```

```

        vim.opt_local.syntax = "off"
        vim.opt_local.filetype = ""
        vim.opt_local.undolevels = -1
        vim.opt_local.swapfile = false
        vim.opt_local.loadplugins = false
    end
end,
})

```

Migration from Vim

Compatibility Layer

```

-- ~/.config/nvim/lua/vim-compat.lua

-- Vim compatibility settings
vim.opt.compatible = false
vim.opt.encoding = 'utf-8'
vim.opt.fileencoding = 'utf-8'

-- Source existing vimrc if needed
local vimrc = vim.fn.expand("~/vimrc")
if vim.fn.filereadable(vimrc) == 1 then
    vim.cmd("source " .. vimrc)
end

-- Convert common vim settings to lua
local function convert_vim_settings()
    -- Add your vim-to-lua conversions here
    if vim.g.loaded_vimrc then
        -- Convert specific vim settings
        vim.opt.number = vim.g.vim_number or false
        vim.opt.relativenumber = vim.g.vim_relativenumber or false
    end
end

convert_vim_settings()

```

Troubleshooting

Common Issues

```
-- ~/.config/nvim/lua/troubleshooting.lua

-- Health check
vim.keymap.set('n', '<leader>h', ':checkhealth<CR>')

-- Plugin debugging
vim.keymap.set('n', '<leader>pi', ':PackerInstall<CR>')
vim.keymap.set('n', '<leader>ps', ':PackerSync<CR>')
vim.keymap.set('n', '<leader>pc', ':PackerClean<CR>')

-- LSP debugging
vim.keymap.set('n', '<leader>li', ':LspInfo<CR>')
vim.keymap.set('n', '<leader>lr', ':LspRestart<CR>')

-- Log files
-- ~/.local/share/nvim/lsp.log
-- ~/.cache/nvim/packer.nvim.log
```

Neovim represents the future of modal editing, combining Vim's efficiency with modern development tools and practices. Its built-in LSP support, Lua scripting, and active plugin ecosystem make it an excellent choice for developers seeking a powerful, customizable editor. ## Next steps

- [LazyVim](#) — batteries-included Neovim
- [tmux](#) · [Herd](#) — session layers
- [Workflows](#) — compose a daily stack

Nano

Nano: The User-Friendly Terminal Editor

Nano is a simple, user-friendly text editor for Unix and Linux systems. Created as a free replacement for the Pico editor, nano provides an intuitive interface with on-screen help, making it ideal for beginners and quick edits.

Why Choose Nano?

- **Beginner-friendly:** Easy to learn with on-screen help
- **No modes:** Direct typing without modal complexity
- **Lightweight:** Minimal resource usage, perfect for Linux servers
- **Universal:** Available on all Linux distributions by default
- **Quick edits:** Perfect for Linux configuration files and system administration
- **Syntax highlighting:** Built-in support for many file types
- **System administration:** Ideal for editing Linux config files and scripts

Installation

Most Linux Distributions (Pre-installed)

```
# Check if nano is installed
which nano
nano --version

# If not installed:
# Ubuntu/Debian
sudo apt update
sudo apt install nano

# CentOS/RHEL/Fedora
sudo yum install nano
# or
sudo dnf install nano

# Arch Linux
sudo pacman -S nano

# Alpine Linux
```

```
sudo apk add nano
```

Additional Linux Distributions

```
# SUSE/openSUSE
sudo zypper install nano

# Gentoo
sudo emerge app-editors/nano

# Void Linux
sudo xbps-install nano
```

Basic Usage

Starting Nano

```
# Create new file
nano filename.txt

# Edit existing file
nano /path/to/file.txt

# Open file at specific line
nano +25 filename.txt

# Open file as read-only
nano -v filename.txt

# Open with line numbers
nano -l filename.txt

# Open with syntax highlighting disabled
nano -Y no filename.txt
```

Interface Overview

GNU nano 6.2

filename.txt

Modified

This is the content of your file.
You can type directly here.
No modes to worry about!

<code>^G</code> Get Help	<code>^O</code> Write Out	<code>^W</code> Where Is	<code>^K</code> Cut Text	<code>^J</code> Justify
<code>^X</code> Exit	<code>^R</code> Read File	<code>^\</code> Replace	<code>^U</code> Paste Text	<code>^T</code> To Spell

The bottom shows available commands where `^` means Ctrl key.

Essential Commands

File Operations

```
# Save file
Ctrl+O (Write Out)
# Then press Enter to confirm filename

# Save as different name
Ctrl+O
# Type new filename and press Enter

# Exit nano
Ctrl+X
# If unsaved changes, nano will ask to save

# Read/Insert another file
Ctrl+R
# Type filename to insert at cursor position

# New buffer (new file)
Alt+F (if multiple buffers enabled)
```

Navigation

```
# Basic movement
Arrow keys      # Move cursor
Ctrl+A          # Beginning of line
Ctrl+E          # End of line
Ctrl+Y          # Previous page
Ctrl+V          # Next page

# Advanced navigation
Ctrl+_          # Go to line number
Ctrl+W          # Search (Where Is)
Alt+W           # Search next occurrence
Ctrl+\  
# Search and replace
```

```
# Word movement
Ctrl+Space      # Move forward one word
Alt+Space       # Move backward one word

# Beginning/End of file
Alt+\           # Go to beginning of file
Alt+/           # Go to end of file
```

Text Editing

```
# Cut and paste
Ctrl+K          # Cut current line
Ctrl+U          # Paste (uncut)
Alt+6           # Copy current line
Ctrl+6          # Mark text (start selection)
                # Move cursor to select text
Ctrl+K          # Cut selected text

# Delete operations
Backspace       # Delete character before cursor
Delete          # Delete character at cursor
Alt+Backspace   # Delete word to the left
Ctrl+Delete     # Delete word to the right

# Undo/Redo
Alt+U           # Undo
Alt+E           # Redo
```

Search and Replace

```
# Search
Ctrl+W          # Open search prompt
# Type search term and press Enter
# Use Ctrl+W again to search for next occurrence

# Search and replace
Ctrl+\          # Open replace prompt
# Enter search term, press Enter
# Enter replacement term, press Enter
# Choose: Y (yes), N (no), A (all), Ctrl+C (cancel)

# Case-sensitive search
Alt+C           # Toggle case sensitivity during search
```

```
# Regular expression search
Alt+R          # Toggle regex mode during search
```

Configuration

Global Configuration

```
# System-wide configuration
/etc/nanorc

# User configuration
~/.nanorc
```

Sample .nanorc Configuration

```
# ~/.nanorc

## Use auto-indentation
set autoindent

## Use bold text instead of reverse video text
set boldtext

## Set the characters treated as closing brackets
set brackets "\>]}}"

## Do case-sensitive searches by default
set casesensitive

## Constantly display the cursor position in the statusbar
set constantshow

## Use cut-from-cursor-to-end-of-line by default
set cutfromcursor

## Set the line length for wrapping text and justifying paragraphs
set fill 72

## Remember the used search/replace strings for the next session
set historylog

## Display line numbers to the left of the text
set linenumbers
```

```

## Enable vim-style lock-files
set locking

## Don't display the helpful shortcut lists at the bottom of the screen
# set nohelp

## Don't automatically add a newline when a file doesn't end with one
set nonewlines

## Set operating directory (chroot of sorts)
# set operatingdir "/tmp"

## Remember the cursor position in each file for the next editing session
set positionlog

## Do extended regular expression searches by default
set regexp

## Make the Home key smarter
set smarthome

## Use smooth scrolling as the default
set smooth

## Use this spelling checker instead of the internal one
set speller "aspell -x -c"

## Allow nano to be suspended
set suspend

## Use this tab size instead of the default; it must be greater than 0
set tabsize 4

## Convert typed tabs to spaces
set tabstospaces

## Save automatically on exit; don't prompt
# set tempfile

## Disallow file modification; why would you want this in an rc file?
# set view

## The two single-column characters used to display the first characters
## of tabs and spaces
set whitespace "»."

```

```
## Detect word boundaries more accurately by treating punctuation
## characters as parts of words
set wordbounds

## Color syntax highlighting
include "/usr/share/nano/*.nanorc"
```

Syntax Highlighting

```
# Enable syntax highlighting for specific file types
# Add to ~/.nanorc:

## Python
include "/usr/share/nano/python.nanorc"

## HTML
include "/usr/share/nano/html.nanorc"

## CSS
include "/usr/share/nano/css.nanorc"

## JavaScript
include "/usr/share/nano/javascript.nanorc"

## JSON
include "/usr/share/nano/json.nanorc"

## Markdown
include "/usr/share/nano/markdown.nanorc"

## Shell scripts
include "/usr/share/nano/sh.nanorc"

## Configuration files
include "/usr/share/nano/conf.nanorc"

## Or include all available syntax files
include "/usr/share/nano/*.nanorc"
```

Custom Syntax Highlighting

```
# ~/.config/nano/custom.nanorc

## Custom syntax for log files
syntax "log" "\.log$"
color red "ERROR.*"
color yellow "WARNING.*"
color green "INFO.*"
color blue "DEBUG.*"
color magenta "\[[0-9-]+\ [0-9:]+\]"

## Custom syntax for configuration files
syntax "config" "\.(conf|config|cfg)$"
color green "^[:space:]*\[^\]=*"
color red "^[:space:]*#.*"
color yellow "\"\[^\"]*"
color cyan "[0-9]+"
```

Advanced Features

Multiple Buffers

```
# Enable multiple buffers in ~/.nanorc
set multibuffer

# Commands for multiple buffers
Alt+F          # Switch to next buffer
Alt+B          # Switch to previous buffer
Ctrl+R         # Read file into new buffer
```

Spell Checking

```
# Install spell checker
sudo apt install aspell aspell-en

# Use spell check in nano
Ctrl+T          # Start spell check
# Navigate through misspelled words
# Choose replacement or skip
```


Backup Files

```
# Enable backup files in ~/.nanorc
set backup

# Set backup directory
set backupdir "/home/user/.nano/backups"

# Create backup directory
mkdir -p ~/.nano/backups
```

Mouse Support

```
# Enable mouse support in ~/.nanorc
set mouse

# Mouse functions:
# - Click to position cursor
# - Double-click to select word
# - Triple-click to select line
# - Scroll wheel to scroll text
```

Real-World Usage Scenarios

Scenario 1: Quick Configuration File Edit

```
# Edit SSH configuration
sudo nano /etc/ssh/sshd_config

# Navigate to specific setting
Ctrl+W
# Search for "PasswordAuthentication"

# Change the value
# Move cursor to "yes" and change to "no"

# Save and exit
Ctrl+O
Enter
Ctrl+X

# Restart SSH service
sudo systemctl restart sshd
```

Scenario 2: System Log Analysis

```
# View system log
sudo nano -v /var/log/syslog

# Search for errors
Ctrl+W
# Type "error" and press Enter

# Continue searching
Ctrl+W
Enter (to search for same term)

# Go to specific time
Ctrl+W
# Type timestamp like "2023-12-01 10:"

# Exit without changes
Ctrl+X
```

Scenario 3: Script Development

```
# Create new script
nano backup_script.sh

# Add shebang and content
#!/bin/bash
# Backup script
DATE=$(date +%Y%m%d)
tar -czf backup_${DATE}.tar.gz /home/user/documents

# Enable line numbers for debugging
Alt+N

# Save and make executable
Ctrl+O
Enter
Ctrl+X
chmod +x backup_script.sh
```

Scenario 4: Crontab Editing

```
# Edit user crontab
crontab -e
# If nano is default editor, it opens in nano

# Add scheduled job
# 0 2 * * * /home/user/backup_script.sh

# Save crontab
Ctrl+O
Enter
Ctrl+X
```

Command Line Options

Useful Nano Options

```
# Open with line numbers
nano -l filename.txt

# Open in restricted mode (no file operations)
nano -R filename.txt

# Set tab width
nano -T 8 filename.txt

# Enable soft wrapping
nano -S filename.txt

# Disable wrapping
nano -w filename.txt

# Show cursor position
nano -c filename.txt

# Enable mouse support
nano -m filename.txt

# Ignore nanorc files
nano -I filename.txt

# Set operating directory
nano -o /tmp filename.txt
```

```
# Enable suspension
nano -z filename.txt
```

Environment Variables

```
# Set nano as default editor
export EDITOR=nano
export VISUAL=nano

# Add to ~/.bashrc or ~/.zshrc
echo 'export EDITOR=nano' >> ~/.bashrc
echo 'export VISUAL=nano' >> ~/.bashrc
```

Tips and Best Practices

Productivity Tips

1. **Learn the shortcuts:** Memorize Ctrl+O (save), Ctrl+X (exit), Ctrl+W (search)
2. **Use line numbers:** Enable with -l option or set linenumbers in .nanorc
3. **Configure syntax highlighting:** Include language-specific nanorc files
4. **Use search effectively:** Ctrl+W for quick navigation in large files
5. **Enable mouse support:** Makes selection and positioning easier

Common Shortcuts Summary

```
# File operations
Ctrl+O      # Save (Write Out)
Ctrl+X      # Exit
Ctrl+R      # Read/Insert file

# Navigation
Ctrl+A      # Beginning of line
Ctrl+E      # End of line
Ctrl+Y      # Previous page
Ctrl+V      # Next page
Ctrl+_      # Go to line number

# Editing
Ctrl+K      # Cut line
Ctrl+U      # Paste (Uncut)
Ctrl+6      # Mark text
Alt+U       # Undo
Alt+E       # Redo
```

```
# Search
Ctrl+W          # Search
Ctrl+\          # Replace
Alt+W           # Find next

# Help
Ctrl+G          # Get help
```

Customization for Different Use Cases

For Programming

```
# ~/.nanorc for programming
set autoindent
set linenumbers
set tabsize 4
set tabstospaces
set mouse
set smooth
set constantshow
include "/usr/share/nano/*.nanorc"
```

For System Administration

```
# ~/.nanorc for sysadmin
set backup
set backupdir "/home/user/.nano/backups"
set linenumbers
set constantshow
set mouse
set locking
include "/usr/share/nano/*.nanorc"
```

For Writing

```
# ~/.nanorc for writing
set fill 80
set justify
set smooth
set mouse
set speller "aspell -x -c"
set tempfile
```

Troubleshooting

Common Issues

1. Nano not found

```
# Install nano
sudo apt install nano
```

2. No syntax highlighting

```
# Check if nanorc files exist
ls /usr/share/nano/
# Add to ~/.nanorc
include "/usr/share/nano/*.nanorc"
```

3. Can't save file (permission denied)

```
# Use sudo
sudo nano /etc/hosts
# Or change file permissions
sudo chmod 644 filename.txt
```

4. Accidental changes

```
# Exit without saving
Ctrl+X
# Choose 'N' when asked to save
```

5. Lost in large file

```
# Go to specific line
Ctrl+_
# Enter line number
```

Recovery Options

```
# If nano crashes, look for backup files
ls ~/.nano/backups/
# or
ls /tmp/

# Recover from backup
nano ~/.nano/backups/filename.txt~
```

Nano excels as a straightforward, accessible text editor that doesn't require learning complex commands or modes. It's perfect for quick edits, system administration tasks, and users who prefer a simple, intuitive interface over the power and complexity of vim or emacs.

Emacs

Emacs: The Extensible Editor

GNU Emacs is one of the most powerful and extensible text editors available. Created by Richard Stallman in 1976, Emacs is more than just an editor—it's a complete computing environment that can be customized and extended using Emacs Lisp.

Why Choose Emacs?

- **Extreme extensibility:** Customize everything with Emacs Lisp
- **Integrated environment:** Email, calendar, file manager, shell, and more
- **Powerful editing:** Advanced text manipulation and programming features
- **Linux integration:** Deep integration with Linux development workflows
- **Active community:** Thousands of packages and configurations
- **Self-documenting:** Comprehensive built-in help system
- **Org-mode:** Powerful organization and note-taking system
- **System administration:** Excellent for managing Linux servers and configurations

Installation

Linux Distributions

```
# Ubuntu/Debian
sudo apt update
sudo apt install emacs

# For GUI version
sudo apt install emacs-gtk

# CentOS/RHEL/Fedora
sudo yum install emacs
# or
sudo dnf install emacs

# Arch Linux
sudo pacman -S emacs

# Alpine Linux
sudo apk add emacs
```

Additional Linux Distributions

```
# SUSE/openSUSE
sudo zypper install emacs

# Gentoo
sudo emerge app-editors/emacs

# Void Linux
sudo xbps-install emacs

# For GUI version on Linux
sudo apt install emacs-gtk      # Ubuntu/Debian with GTK
sudo pacman -S emacs           # Arch Linux (includes GUI)
```

Building from Source

```
# Install dependencies (Ubuntu/Debian)
sudo apt install build-essential texinfo libx11-dev libxpm-dev \
    libjpeg-dev libpng-dev libgif-dev libtiff-dev libgtk-3-dev \
    libncurses-dev libxpm-dev automake autoconf

# Download and build
wget https://ftp.gnu.org/gnu/emacs/emacs-29.1.tar.xz
tar -xJf emacs-29.1.tar.xz
cd emacs-29.1
./configure --with-native-compilation
make -j$(nproc)
sudo make install
```

Basic Concepts

Key Notation

- C-x means Ctrl+x
- M-x means Alt+x (Meta key)
- C-x C-f means Ctrl+x followed by Ctrl+f
- RET means Enter key
- SPC means Space key

Buffers, Windows, and Frames

- **Buffer:** A file or content in memory

- **Window:** A view of a buffer (can have multiple windows)
- **Frame:** The entire Emacs window (can have multiple frames)

Major and Minor Modes

- **Major Mode:** Defines the primary editing behavior (e.g., Python mode, HTML mode)
- **Minor Mode:** Additional features that can be enabled/disabled (e.g., line numbers, auto-complete)

Essential Commands

Starting and Exiting

```
# Start Emacs
emacs

# Start in terminal mode
emacs -nw

# Open specific file
emacs filename.txt

# Exit Emacs
C-x C-c
```

File Operations

```
# Open file
C-x C-f
# Type filename and press RET

# Save file
C-x C-s

# Save as (write file)
C-x C-w

# Save all buffers
C-x s

# Insert file contents
C-x i
```

```
# Recent files
C-x C-r
```

Buffer Management

```
# Switch buffer
C-x b
# Type buffer name or use TAB completion

# List all buffers
C-x C-b

# Kill (close) buffer
C-x k

# Switch to next buffer
C-x →

# Switch to previous buffer
C-x ←
```

Window Management

```
# Split window horizontally
C-x 2

# Split window vertically
C-x 3

# Switch to other window
C-x o

# Delete current window
C-x 0

# Delete other windows
C-x 1

# Resize windows
C-x ^    # Make taller
C-x }    # Make wider
C-x {    # Make narrower
```

Navigation and Editing

Basic Movement

```
# Character movement
C-f      # Forward character
C-b      # Backward character
C-n      # Next line
C-p      # Previous line

# Word movement
M-f      # Forward word
M-b      # Backward word

# Line movement
C-a      # Beginning of line
C-e      # End of line

# Sentence movement
M-a      # Beginning of sentence
M-e      # End of sentence

# Paragraph movement
M-{       # Beginning of paragraph
M-}       # End of paragraph

# Page movement
C-v      # Page down
M-v      # Page up

# Buffer movement
M-<       # Beginning of buffer
M->       # End of buffer
```

Advanced Navigation

```
# Go to line
M-g g    # Go to line number
M-g M-g  # Alternative

# Go to character
M-g c    # Go to character position

# Jump to definition
M-.      # Find definition
```

```
M-,      # Return from definition
```

```
# Bookmarks
```

```
C-x r m  # Set bookmark
```

```
C-x r b  # Jump to bookmark
```

```
C-x r l  # List bookmarks
```

Text Editing

```
# Delete operations
```

```
C-d      # Delete character forward
```

```
DEL      # Delete character backward
```

```
M-d      # Delete word forward
```

```
M-DEL    # Delete word backward
```

```
C-k      # Kill to end of line
```

```
M-k      # Kill to end of sentence
```

```
# Kill and yank (cut and paste)
```

```
C-w      # Kill region
```

```
M-w      # Copy region
```

```
C-y      # Yank (paste)
```

```
M-y      # Cycle through kill ring
```

```
# Undo
```

```
C-/      # Undo
```

```
C-x u    # Undo
```

```
C-g      # Quit/cancel command
```

Selection (Regions)

```
# Set mark
```

```
C-SPC    # Set mark at cursor
```

```
C-x C-x  # Exchange point and mark
```

```
# Select all
```

```
C-x h    # Select entire buffer
```

```
# Rectangle selection
```

```
C-x r r  # Copy rectangle
```

```
C-x r k  # Kill rectangle
```

```
C-x r y  # Yank rectangle
```

```
C-x r t  # Replace rectangle with text
```

Search and Replace

Basic Search

```
# Incremental search forward
C-s
# Type search term, C-s for next occurrence

# Incremental search backward
C-r

# Word search
C-s C-w # Search for word at cursor

# Regular expression search
C-M-s    # Forward regex search
C-M-r    # Backward regex search
```

Replace Operations

```
# Query replace
M-%
# Enter search term, then replacement
# y (yes), n (no), ! (all), q (quit)

# Regular expression replace
C-M-%

# Replace in region
# Select region first, then M-%
```

Advanced Search

```
# Occur (show all lines matching pattern)
M-s o

# Multi-occur (search in multiple buffers)
M-s M-o

# Grep in files
M-x grep
M-x rgrep    # Recursive grep
```

Configuration

Init File

Emacs configuration is stored in `~/.emacs.d/init.el` or `~/.emacs`:

```
;; ~/.emacs.d/init.el - Basic configuration

;; Package management
(require 'package)
(setq package-archives '(("melpa" . "https://melpa.org/packages/")
                        ("org" . "https://orgmode.org/elpa/")
                        ("elpa" . "https://elpa.gnu.org/packages/")))
(package-initialize)

;; Bootstrap use-package
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package))

(eval-when-compile
  (require 'use-package))

;; Basic settings
(setq inhibit-startup-message t)          ; Disable startup screen
(setq visible-bell t)                     ; Visual bell instead of beep
(setq make-backup-files nil)              ; Disable backup files
(setq auto-save-default nil)              ; Disable auto-save

;; UI improvements
(tool-bar-mode -1)                       ; Disable toolbar
(menu-bar-mode -1)                       ; Disable menu bar
(scroll-bar-mode -1)                     ; Disable scroll bar
(global-display-line-numbers-mode 1)     ; Show line numbers
(column-number-mode 1)                   ; Show column numbers
(show-paren-mode 1)                      ; Highlight matching parentheses

;; Indentation
(setq-default indent-tabs-mode nil)       ; Use spaces instead of tabs
(setq-default tab-width 4)                ; Tab width
(setq indent-line-function 'insert-tab)   ; Tab behavior

;; Font and theme
(set-face-attribute 'default nil :font "Fira Code-12")
(load-theme 'wombat t)

;; Key bindings
```



```

(global-set-key (kbd "C-x C-b") 'ibuffer)
(global-set-key (kbd "M-/") 'hippie-expand)
(global-set-key (kbd "C-c c") 'comment-or-uncomment-region)

;; Auto-completion
(use-package company
  :ensure t
  :init
  (global-company-mode 1))

;; Syntax checking
(use-package flycheck
  :ensure t
  :init
  (global-flycheck-mode))

;; Git integration
(use-package magit
  :ensure t
  :bind ("C-x g" . magit-status))

;; Project management
(use-package projectile
  :ensure t
  :init
  (projectile-mode +1)
  :bind-keymap
  ("C-c p" . projectile-command-map))

;; File tree
(use-package neotree
  :ensure t
  :bind ("F8" . neotree-toggle))

;; Better search
(use-package ivy
  :ensure t
  :init
  (ivy-mode 1)
  :config
  (setq ivy-use-virtual-buffers t)
  (setq enable-recursive-minibuffers t))

(use-package counsel
  :ensure t
  :bind (("M-x" . counsel-M-x)
        ("C-x C-f" . counsel-find-file))

```

```

        ("C-h f" . counsel-describe-function)
        ("C-h v" . counsel-describe-variable)))

(use-package swiper
  :ensure t
  :bind ("C-s" . swiper))

```

Package Management

```

;; Install packages manually
M-x package-install RET package-name RET

;; List packages
M-x list-packages

;; Update packages
M-x package-list-packages
# Press 'U' then 'x' to update all

```

Popular Packages

```

;; Essential packages configuration

;; Helm (alternative to ivy/counsel)
(use-package helm
  :ensure t
  :bind (("M-x" . helm-M-x)
        ("C-x C-f" . helm-find-files)
        ("C-x b" . helm-buffers-list))
  :config
  (helm-mode 1))

;; Which-key (show key bindings)
(use-package which-key
  :ensure t
  :config
  (which-key-mode))

;; Multiple cursors
(use-package multiple-cursors
  :ensure t
  :bind (("C-S-c C-S-c" . mc/edit-lines)
        ("C->" . mc/mark-next-like-this))

```

```

        ("C-<" . mc/mark-previous-like-this)))

;; Expand region
(use-package expand-region
  :ensure t
  :bind ("C= " . er/expand-region))

;; Yasnippet (snippets)
(use-package yasnippet
  :ensure t
  :config
  (yas-global-mode 1))

;; Language Server Protocol
(use-package lsp-mode
  :ensure t
  :hook ((python-mode . lsp)
        (js-mode . lsp)
        (typescript-mode . lsp))
  :commands lsp)

(use-package lsp-ui
  :ensure t
  :commands lsp-ui-mode)

;; Company LSP integration
(use-package company-lsp
  :ensure t
  :commands company-lsp)

```

Programming with Emacs

Language-Specific Modes

```

;; Python development
(use-package python-mode
  :ensure t
  :mode "\\\\.py\\\\"
  :config
  (setq python-indent-offset 4))

;; JavaScript/TypeScript
(use-package js2-mode
  :ensure t

```

```

:mode "\\js\\'"
:config
(setq js2-basic-offset 2))

(use-package typescript-mode
  :ensure t
  :mode "\\ts\\'")

;; Web development
(use-package web-mode
  :ensure t
  :mode ("\\.html\\'" "\\.css\\'" "\\.jsx\\'")
  :config
  (setq web-mode-markup-indent-offset 2)
  (setq web-mode-css-indent-offset 2)
  (setq web-mode-code-indent-offset 2))

;; Markdown
(use-package markdown-mode
  :ensure t
  :mode "\\md\\'")

;; YAML
(use-package yaml-mode
  :ensure t
  :mode "\\ya?ml\\'")

;; JSON
(use-package json-mode
  :ensure t
  :mode "\\json\\'")

```

Code Navigation

```

# Tags (requires etags or ctags)
M-.      # Find definition
M-,      # Return from definition
M-*      # Pop tag mark

# Imenu (function/class navigation)
M-x imenu

# Occur (find all occurrences)
M-s o

```

```
# Grep
M-x grep
M-x rgrep
```

Debugging

```
;; GDB integration
M-x gdb

;; Python debugging with pdb
M-x pdb

;; Edebug (Emacs Lisp debugger)
C-u C-M-x ; Instrument function for debugging
```

Org Mode

Org-mode is one of Emacs' most powerful features for organization and note-taking:

Basic Org Syntax

```
* Top Level Heading
** Second Level Heading
*** Third Level Heading

- Unordered list item
- Another item
  - Sub-item

1. Ordered list item
2. Another ordered item

*bold* /italic/ _underlined_ =code= ~verbatim~

[[https://example.com][Link text]]

#+BEGIN_SRC python
def hello_world():
    print("Hello, World!")
#+END_SRC

| Name | Age | City |
```

```
|-----+-----+-----|
| John | 25 | NYC |
| Jane | 30 | LA |
```

Org Mode Commands

```
# Structure editing
TAB      # Cycle visibility
S-TAB    # Global visibility cycling
M-RET    # Insert new heading
M-+/->   # Promote/demote heading
M-↑/↓    # Move heading up/down

# TODO items
C-c C-t  # Cycle TODO states
C-c C-s  # Schedule item
C-c C-d  # Set deadline

# Tables
C-c |    # Create table
TAB      # Move to next field
S-TAB    # Move to previous field
C-c C-c  # Align table

# Links
C-c C-l  # Insert link
C-c C-o  # Open link

# Export
C-c C-e  # Export menu
```

Org Configuration

```
;; Org-mode configuration
(use-package org
  :ensure t
  :config
  (setq org-todo-keywords
        '((sequence "TODO" "IN-PROGRESS" "WAITING" "|" "DONE" "CANCELLED"))))
  (setq org-agenda-files '("~/org/"))
  (setq org-default-notes-file "~/org/notes.org")
  :bind (("C-c l" . org-store-link)
         ("C-c a" . org-agenda)
         ("C-c c" . org-capture)))
```

```
;; Org capture templates
(setq org-capture-templates
  '(("t" "Todo" entry (file+headline "~/org/tasks.org" "Tasks")
    "* TODO %?\n %i\n %a")
    ("n" "Note" entry (file+headline "~/org/notes.org" "Notes")
    "* %?\nEntered on %U\n %i\n %a"))))
```

Real-World Development Scenarios

Scenario 1: Python Development Setup

```
;; Python development configuration
(use-package python-mode
  :ensure t
  :config
  (setq python-indent-offset 4)
  (add-hook 'python-mode-hook
    (lambda ()
      (setq indent-tabs-mode nil)
      (setq tab-width 4)
      (setq python-indent-offset 4))))

;; Virtual environment support
(use-package pyvenv
  :ensure t
  :config
  (pyvenv-mode 1))

;; Python testing
(use-package python-pytest
  :ensure t
  :bind (:map python-mode-map
    ("C-c t" . python-pytest)))

;; Jupyter notebooks
(use-package ein
  :ensure t)
```

Scenario 2: Web Development Workflow

```
;; Web development setup
(use-package web-mode
  :ensure t
  :mode ("\\.html\\'" "\\.css\\'" "\\.js\\'" "\\.jsx\\'" "\\.tsx\\'")
  :config
  (setq web-mode-markup-indent-offset 2)
  (setq web-mode-css-indent-offset 2)
  (setq web-mode-code-indent-offset 2)
  (setq web-mode-enable-auto-pairing t)
  (setq web-mode-enable-css-colorization t))

;; Emmet for HTML expansion
(use-package emmet-mode
  :ensure t
  :hook (web-mode . emmet-mode))

;; Live browser reload
(use-package impatient-mode
  :ensure t)

;; REST client
(use-package restclient
  :ensure t
  :mode ("\\.http\\'" . restclient-mode))
```

Scenario 3: System Administration

```
;; System administration tools
(use-package tramp
  :config
  (setq tramp-default-method "ssh"))

;; Docker integration
(use-package docker
  :ensure t
  :bind ("C-c d" . docker))

(use-package dockerfile-mode
  :ensure t
  :mode "Dockerfile\\'")

;; Ansible
(use-package ansible
```



```

:ensure t)

;; Log file viewing
(use-package logview
  :ensure t
  :mode ("\\.log\\" . logview-mode))

```

Advanced Emacs Features

Macros

```

# Record macro
C-x (    # Start recording
# ... perform actions ...
C-x )    # Stop recording

# Execute macro
C-x e    # Execute once
C-u 10 C-x e # Execute 10 times

# Name and save macro
M-x name-last-kbd-macro
M-x insert-kbd-macro

```

Registers

```

# Save to register
C-x r s a    # Save region to register 'a'
C-x r i a    # Insert from register 'a'

# Save position
C-x r SPC a  # Save position to register 'a'
C-x r j a    # Jump to register 'a'

# Save window configuration
C-x r w a    # Save window config to register 'a'
C-x r j a    # Restore window config from register 'a'

```

Dired (Directory Editor)

```
# Open dired
C-x d

# Dired commands
RET      # Open file/directory
q        # Quit dired
g        # Refresh
m        # Mark file
u        # Unmark file
d        # Mark for deletion
x        # Execute deletions
C        # Copy file
R        # Rename/move file
+        # Create directory
```

Shell Integration

```
# Shell commands
M-!      # Execute shell command
M-|      # Shell command on region
C-u M-!  # Insert command output

# Shell mode
M-x shell
M-x eshell # Emacs shell
M-x term   # Terminal emulator
```

Customization and Themes

Custom Themes

```
;; Popular themes
(use-package doom-themes
  :ensure t
  :config
  (load-theme 'doom-one t))

(use-package solarized-theme
  :ensure t)

(use-package zenburn-theme
```

```

:ensure t)

;; Theme switching function
(defun switch-theme (theme)
  "Disable current theme and load THEME."
  (interactive
   (list
    (intern (completing-read "Load custom theme: "
                           (mapcar 'symbol-name
                                   (custom-available-themes))))))
  (let ((enabled-themes custom-enabled-themes))
    (mapc #'disable-theme custom-enabled-themes)
    (load-theme theme t)))

```

Custom Functions

```

;; Useful custom functions
(defun duplicate-line ()
  "Duplicate current line."
  (interactive)
  (move-beginning-of-line 1)
  (kill-line)
  (yank)
  (open-line 1)
  (next-line 1)
  (yank))

(global-set-key (kbd "C-c d") 'duplicate-line)

(defun comment-or-uncomment-region-or-line ()
  "Comments or uncomments the region or the current line if there's no active region."
  (interactive)
  (let (beg end)
    (if (region-active-p)
        (setq beg (region-beginning) end (region-end))
        (setq beg (line-beginning-position) end (line-end-position)))
    (comment-or-uncomment-region beg end)))

(global-set-key (kbd "C-/") 'comment-or-uncomment-region-or-line)

```

Performance Optimization

Startup Optimization

```
;; Startup optimization
(setq gc-cons-threshold (* 50 1000 1000)) ; Increase GC threshold
(setq read-process-output-max (* 1024 1024)) ; Increase read buffer

;; Restore GC threshold after startup
(add-hook 'emacs-startup-hook
  (lambda ()
    (setq gc-cons-threshold (* 2 1000 1000))))

;; Lazy loading
(use-package package-name
  :ensure t
  :defer t ; Lazy load
  :hook (mode . package-name))
```

Large File Handling

```
;; Large file handling
(defun my-find-file-check-make-large-file-read-only-hook ()
  "If a file is over a given size, make the buffer read only."
  (when (> (buffer-size) (* 1024 1024))
    (setq buffer-read-only t)
    (buffer-disable-undo)
    (fundamental-mode)))

(add-hook 'find-file-hook 'my-find-file-check-make-large-file-read-only-hook)
```

Troubleshooting

Common Issues

```
# Debug init file
emacs --debug-init

# Start with minimal config
emacs -Q

# Profile startup time
M-x emacs-init-time
```

```
# Check package issues
M-x package-list-packages
# Look for packages with issues

# Reset package system
# Delete ~/.emacs.d/elpa/ and restart
```

Recovery

```
# Auto-save recovery
M-x recover-file

# Session recovery
M-x desktop-save-mode ; Enable session saving
M-x desktop-read      ; Restore session
```

Emacs is incredibly powerful and customizable, making it suitable for everything from simple text editing to complex development workflows. While it has a steep learning curve, the investment pays off with a highly personalized and efficient editing environment.

LazyVim

LazyVim: Modern Neovim Configuration

LazyVim is a popular, pre-configured Neovim setup that provides a modern IDE-like experience out of the box. Built on top of lazy.nvim plugin manager, it offers sensible defaults, beautiful UI, and powerful features for developers.

What is LazyVim?

LazyVim is a Neovim configuration framework that includes: - **Pre-configured plugins**: Carefully selected and configured plugins - **Lazy loading**: Fast startup times with lazy.nvim - **Modern UI**: Beautiful interface with proper theming - **LSP integration**: Built-in Language Server Protocol support - **Extensible**: Easy to customize and extend - **Well-documented**: Comprehensive documentation and examples

Why Choose LazyVim?

- **Zero configuration**: Works great out of the box
- **Fast startup**: Optimized for performance
- **Modern features**: Latest Neovim capabilities
- **Active development**: Regular updates and improvements
- **Great defaults**: Sensible keybindings and settings
- **Easy customization**: Simple configuration structure

Installation

Prerequisites

```
# Neovim 0.9.0 or higher
nvim --version

# Install Neovim if needed
# Ubuntu/Debian
sudo apt install neovim

# Arch Linux
sudo pacman -S neovim
```

```
# SUSE/openSUSE
sudo zypper install neovim

# Gentoo
sudo emerge app-editors/neovim
```

Required Dependencies

```
# Git (required)
git --version

# Node.js (for LSP servers)
node --version
npm --version

# Python (for some plugins)
python3 --version
pip3 --version

# Ripgrep (for telescope)
rg --version

# fd (for telescope)
fd --version

# Install missing dependencies
# Ubuntu/Debian
sudo apt install git nodejs npm python3 python3-pip ripgrep fd-find

# Arch Linux
sudo pacman -S git nodejs npm python python-pip ripgrep fd

# SUSE/openSUSE
sudo zypper install git nodejs npm python3 python3-pip ripgrep fd

# Gentoo
sudo emerge dev-vcs/git net-libs/nodejs dev-lang/python sys-apps/ripgrep sys-apps/fd
```

Installation Steps

```
# Backup existing Neovim configuration
mv ~/.config/nvim ~/.config/nvim.bak
mv ~/.local/share/nvim ~/.local/share/nvim.bak
```



```

mv ~/.local/state/nvim ~/.local/state/nvim.bak
mv ~/.cache/nvim ~/.cache/nvim.bak

# Clone LazyVim starter template
git clone https://github.com/LazyVim/starter ~/.config/nvim

# Remove .git folder to make it your own
rm -rf ~/.config/nvim/.git

# Start Neovim (plugins will install automatically)
nvim

```

Initial Setup and Configuration

First Launch

When you first start LazyVim:

1. **Plugin Installation:** Plugins install automatically
2. **Mason Setup:** Language servers install automatically
3. **Treesitter:** Syntax parsers download automatically
4. **Health Check:** Run `:checkhealth` to verify setup

Basic Configuration Structure

```

~/.config/nvim/
  init.lua           # Entry point
  lua/
    config/
      autocmds.lua   # Auto commands
      keymaps.lua    # Key mappings
      lazy.lua        # Plugin manager setup
      options.lua     # Neovim options
    plugins/
      example.lua     # Example plugin config
      ...             # Your plugin configurations
  stylua.toml        # Lua formatter config

```

Basic Options Configuration

```

-- ~/.config/nvim/lua/config/options.lua

-- LazyVim auto format
vim.g.autoformat = true

```

```

-- LazyVim picker to use
-- Can be one of: telescope, fzf
vim.g.lazyvim_picker = "telescope"

-- LazyVim root dir detection
-- Each entry can be:
-- * the name of a detector function like `lsp` or `cwd`
-- * a pattern or array of patterns like `.git` or `lua`.
-- * a function with signature `function(buf) -> string|string[]`
vim.g.root_spec = { "lsp", { ".git", "lua" }, "cwd" }

-- LazyVim automatically configures lazygit:
-- * theme, based on the active colorscheme.
-- * editorPreset to nvim-remote
-- * enables nerd font icons
-- Set to false to disable.
vim.g.lazygit_config = true

-- Options are automatically loaded before lazy.nvim startup
-- Default options that are always set: https://github.com/LazyVim/LazyVim/blob/main/lua/laz
-- Add any additional options here

-- Line numbers
vim.opt.relativenumber = true -- Relative line numbers
vim.opt.number = true -- Print line number

-- Tabs / Indentation
vim.opt.tabstop = 2 -- 2 spaces for tabs (prettier default)
vim.opt.shiftwidth = 2 -- 2 spaces for indent width
vim.opt.expandtab = true -- expand tab to spaces
vim.opt.autoindent = true -- copy indent from current line when starting new one

-- Line wrapping
vim.opt.wrap = false -- disable line wrapping

-- Search settings
vim.opt.ignorecase = true -- ignore case when searching
vim.opt.smartcase = true -- if you include mixed case in your search, assumes you want case-

-- Cursor line
vim.opt.cursorline = true -- highlight the current cursor line

-- Appearance
vim.opt.termguicolors = true
vim.opt.background = "dark" -- colorschemes that can be light or dark will be made dark
vim.opt.signcolumn = "yes" -- show sign column so that text doesn't shift

```

```

-- Backspace
vim.opt.backspace = "indent,eol,start" -- allow backspace on indent, end of line or insert m

-- Clipboard
vim.opt.clipboard:append("unnamedplus") -- use system clipboard as default register

-- Split windows
vim.opt.splitright = true -- split vertical window to the right
vim.opt.splitbelow = true -- split horizontal window to the bottom

-- Turn off swapfile
vim.opt.swapfile = false

```

Custom Keymaps

```

-- ~/.config/nvim/lua/config/keymaps.lua

-- Keymaps are automatically loaded on the VeryLazy event
-- Default keymaps that are always set: https://github.com/LazyVim/LazyVim/blob/main/lua/laz
-- Add any additional keymaps here

local keymap = vim.keymap -- for conciseness

-- General keymaps
keymap.set("i", "jk", "<ESC>", { desc = "Exit insert mode with jk" })

keymap.set("n", "<leader>nh", ":nohl<CR>", { desc = "Clear search highlights" })

-- increment/decrement numbers
keymap.set("n", "<leader>+", "<C-a>", { desc = "Increment number" }) -- increment
keymap.set("n", "<leader>-", "<C-x>", { desc = "Decrement number" }) -- decrement

-- window management
keymap.set("n", "<leader>sv", "<C-w>v", { desc = "Split window vertically" }) -- split windo
keymap.set("n", "<leader>sh", "<C-w>s", { desc = "Split window horizontally" }) -- split win
keymap.set("n", "<leader>se", "<C-w>=", { desc = "Make splits equal size" }) -- make split w
keymap.set("n", "<leader>sx", "<cmd>close<CR>", { desc = "Close current split" }) -- close c

keymap.set("n", "<leader>to", "<cmd>tabnew<CR>", { desc = "Open new tab" }) -- open new tab
keymap.set("n", "<leader>tx", "<cmd>tabclose<CR>", { desc = "Close current tab" }) -- close
keymap.set("n", "<leader>tn", "<cmd>tabn<CR>", { desc = "Go to next tab" }) -- go to next t
keymap.set("n", "<leader>tp", "<cmd>tabp<CR>", { desc = "Go to previous tab" }) -- go to pr
keymap.set("n", "<leader>tf", "<cmd>tabnew %<CR>", { desc = "Open current buffer in new tab" })

-- Move text up and down

```

```

keymap.set("v", "J", ":m '>+1<CR>gv=gv", { desc = "Move text down" })
keymap.set("v", "K", ":m '<-2<CR>gv=gv", { desc = "Move text up" })

-- Stay in indent mode
keymap.set("v", "<", "<gv", { desc = "Indent left" })
keymap.set("v", ">", ">gv", { desc = "Indent right" })

-- Keep last yanked when pasting
keymap.set("v", "p", '"_dP', { desc = "Paste without yanking" })

```

Auto Commands

```

-- ~/.config/nvim/lua/config/autocmds.lua

-- Autocmds are automatically loaded on the VeryLazy event
-- Default autocmds that are always set: https://github.com/LazyVim/LazyVim/blob/main/lua/la
-- Add any additional autocmds here

local function augroup(name)
    return vim.api.nvim_create_augroup("lazyvim_" .. name, { clear = true })
end

-- Highlight on yank
vim.api.nvim_create_autocmd("TextYankPost", {
    group = augroup("highlight_yank"),
    callback = function()
        vim.highlight.on_yank()
    end,
})

-- resize splits if window got resized
vim.api.nvim_create_autocmd({ "VimResized" }, {
    group = augroup("resize_splits"),
    callback = function()
        local current_tab = vim.fn.tabpagenr()
        vim.cmd("tabdo wincmd =")
        vim.cmd("tabnext " .. current_tab)
    end,
})

-- go to last loc when opening a buffer
vim.api.nvim_create_autocmd("BufReadPost", {
    group = augroup("last_loc"),
    callback = function(event)
        local exclude = { "gitcommit" }
    end,
})

```

```

    local buf = event.buf
    if vim.tbl_contains(exclude, vim.bo[buf].filetype) or vim.b[buf].lazyvim_last_loc then
        return
    end
    vim.b[buf].lazyvim_last_loc = true
    local mark = vim.api.nvim_buf_get_mark(buf, '')
    local lcount = vim.api.nvim_buf_line_count(buf)
    if mark[1] > 0 and mark[1] <= lcount then
        pcall(vim.api.nvim_win_set_cursor, 0, mark)
    end
end,
})

-- close some filetypes with <q>
vim.api.nvim_create_autocmd("FileType", {
    group = augroup("close_with_q"),
    pattern = {
        "PlenaryTestPopup",
        "help",
        "lspinfo",
        "man",
        "notify",
        "qf",
        "query",
        "spectre_panel",
        "startuptime",
        "tsplayground",
        "neotest-output",
        "checkhealth",
        "neotest-summary",
        "neotest-output-panel",
    },
    callback = function(event)
        vim.bo[event.buf].buflisted = false
        vim.keymap.set("n", "q", "<cmd>close<cr>", { buffer = event.buf, silent = true })
    end,
})

-- wrap and check for spell in text filetypes
vim.api.nvim_create_autocmd("FileType", {
    group = augroup("wrap_spell"),
    pattern = { "gitcommit", "markdown" },
    callback = function()
        vim.opt_local.wrap = true
        vim.opt_local.spell = true
    end,
})

```

```

-- Auto create dir when saving a file, in case some intermediate directory does not exist
vim.api.nvim_create_autocmd({ "BufWritePre" }, {
  group = augroup("auto_create_dir"),
  callback = function(event)
    if event.match:match("^%w%w+:.//") then
      return
    end
    local file = vim.loop.fs_realpath(event.match) or event.match
    vim.fn.mkdir(vim.fn.fnamemodify(file, ":p:h"), "p")
  end,
})

```

Plugin Configuration

Adding Custom Plugins

```

-- ~/.config/nvim/lua/plugins/example.lua

return {
  -- Add your custom plugins here
  {
    "folke/flash.nvim",
    event = "VeryLazy",
    opts = {},
    keys = {
      {
        "s",
        mode = { "n", "x", "o" },
        function()
          require("flash").jump()
        end,
        desc = "Flash",
      },
      {
        "S",
        mode = { "n", "o", "x" },
        function()
          require("flash").treesitter()
        end,
        desc = "Flash Treesitter",
      },
    },
  },
}

```

```

-- Disable default plugins
{ "folke/flash.nvim", enabled = false },

-- Override plugin configuration
{
  "nvim-telescope/telescope.nvim",
  opts = {
    defaults = {
      layout_strategy = "horizontal",
      layout_config = { prompt_position = "top" },
      sorting_strategy = "ascending",
      winblend = 0,
    },
  },
},
}

```

Language-Specific Configurations

```

-- ~/.config/nvim/lua/plugins/languages.lua

return {
  -- Python
  {
    "nvim-treesitter/nvim-treesitter",
    opts = function(_, opts)
      if type(opts.ensure_installed) == "table" then
        vim.list_extend(opts.ensure_installed, { "python" })
      end
    end,
  },
  {
    "neovim/nvim-lspconfig",
    opts = {
      servers = {
        pyright = {},
        ruff_lsp = {},
      },
    },
  },
  -- JavaScript/TypeScript
  {
    "nvim-treesitter/nvim-treesitter",
    opts = function(_, opts)

```

```

        if type(opts.ensure_installed) == "table" then
            vim.list_extend(opts.ensure_installed, { "javascript", "typescript", "tsx" })
        end
    end,
},
{
    "neovim/nvim-lspconfig",
    opts = {
        servers = {
            tsserver = {},
            eslint = {},
        },
    },
},
-- Go
{
    "nvim-treesitter/nvim-treesitter",
    opts = function(_, opts)
        if type(opts.ensure_installed) == "table" then
            vim.list_extend(opts.ensure_installed, { "go", "gomod", "gowork", "gosum" })
        end
    end,
},
{
    "neovim/nvim-lspconfig",
    opts = {
        servers = {
            gopls = {},
        },
    },
},
-- Rust
{
    "nvim-treesitter/nvim-treesitter",
    opts = function(_, opts)
        if type(opts.ensure_installed) == "table" then
            vim.list_extend(opts.ensure_installed, { "rust", "toml" })
        end
    end,
},
{
    "neovim/nvim-lspconfig",
    opts = {
        servers = {

```



```

        rust_analyzer = {},
    },
},
},
}

```

Essential LazyVim Features

File Explorer (Neo-tree)

```

# Default keymaps
<leader>fe    # Open file explorer
<leader>fE    # Open file explorer (root dir)
<leader>e     # Toggle file explorer
<leader>E     # Toggle file explorer (root dir)

# Neo-tree navigation
j/k          # Move up/down
h/l          # Collapse/expand or enter
<CR>         # Open file
o            # Open file
s            # Open in split
v            # Open in vsplit
t            # Open in tab
a            # Add file/directory
d            # Delete
r            # Rename
c            # Copy
x            # Cut
p            # Paste
q            # Close

```

Fuzzy Finding (Telescope)

```

# File finding
<leader>ff    # Find files
<leader>fr    # Recent files
<leader>fg    # Find files (git)
<leader>fF    # Find files (cwd)

# Text searching
<leader>sg    # Grep search
<leader>sw    # Search word under cursor

```

```

<leader>sG      # Grep search (cwd)
<leader>ss      # Search symbols
<leader>sS      # Search symbols (workspace)

# Buffer management
<leader>fb      # Find buffers
<leader>,       # Switch buffers

# Git integration
<leader>gc      # Git commits
<leader>gs      # Git status

# Help and commands
<leader>sh      # Help pages
<leader>sk      # Key maps
<leader>sc      # Commands
<leader>sC      # Command history

```

LSP Features

```

# Navigation
gd              # Go to definition
gr              # Go to references
gI              # Go to implementation
gy              # Go to type definition
gD              # Go to declaration

# Information
K              # Hover information
gK              # Signature help
<c-k>          # Signature help (insert mode)

# Code actions
<leader>ca      # Code actions
<leader>cA      # Source actions
<leader>cr      # Rename
<leader>cf      # Format
<leader>cF      # Format (injected langs)

# Diagnostics
<leader>cd      # Line diagnostics
<leader>cD      # Workspace diagnostics
]d              # Next diagnostic
[d              # Previous diagnostic
]e              # Next error

```

```
[e          # Previous error
]w          # Next warning
[w          # Previous warning
```

Git Integration (Lazygit)

```
# Lazygit
<leader>gg  # Open Lazygit
<leader>gG  # Open Lazygit (root dir)

# Git signs (gitsigns)
]h          # Next hunk
[h          # Previous hunk
<leader>ghs  # Stage hunk
<leader>ghr  # Reset hunk
<leader>ghS  # Stage buffer
<leader>ghu  # Undo stage hunk
<leader>ghR  # Reset buffer
<leader>ghp  # Preview hunk
<leader>ghb  # Blame line
<leader>ghl  # Blame line (full)
<leader>ghd  # Diff this
<leader>ghD  # Diff this ~
```

Terminal Integration

```
# Terminal
<leader>ft  # Terminal (root dir)
<leader>fT  # Terminal (cwd)
<c- />     # Terminal (root dir)
<c- _>     # Terminal (root dir) (which-key)

# Terminal navigation (in terminal mode)
<esc><esc>  # Enter normal mode
<C-h>      # Go to left window
<C-j>      # Go to lower window
<C-k>      # Go to upper window
<C-l>      # Go to right window
```

Real-World Development Setups

Python Development

```
-- ~/.config/nvim/lua/plugins/python.lua

return {
  -- Python-specific plugins
  {
    "linux-cultist/venv-selector.nvim",
    dependencies = { "neovim/nvim-lspconfig", "nvim-telescope/telescope.nvim", "mfussenegger"
    opts = {
      -- Your options go here
      name = "venv",
      auto_refresh = false,
    },
    event = "VeryLazy", -- Optional: needed only if you want to type `:VenvSelect` without a
    keys = {
      -- Keymap to open VenvSelector to pick a venv.
      { "<leader>vs", "<cmd>VenvSelect<cr>" },
      -- Keymap to retrieve the venv from a cache (the one previously used for the same proj
      { "<leader>vc", "<cmd>VenvSelectCached<cr>" },
    },
  },
},

-- Python debugging
{
  "mfussenegger/nvim-dap-python",
  ft = "python",
  dependencies = {
    "mfussenegger/nvim-dap",
    "rcarriga/nvim-dap-ui",
  },
  config = function(_, opts)
    local path = "~/local/share/nvim/mason/packages/debugpy/venv/bin/python"
    require("dap-python").setup(path)
  end,
},

-- Python testing
{
  "nvim-neotest/neotest",
  optional = true,
  dependencies = {
    "nvim-neotest/neotest-python",
  },
},
```

```

opts = {
  adapters = {
    ["neotest-python"] = {
      -- Here you can specify the settings for the adapter, i.e.
      runner = "pytest",
      python = ".venv/bin/python",
    },
  },
},

-- Enhanced Python syntax
{
  "nvim-treesitter/nvim-treesitter",
  opts = function(_, opts)
    if type(opts.ensure_installed) == "table" then
      vim.list_extend(opts.ensure_installed, { "ninja", "python", "rst", "toml" })
    end
  end,
},

-- Python LSP configuration
{
  "neovim/nvim-lspconfig",
  opts = {
    servers = {
      pyright = {
        settings = {
          python = {
            analysis = {
              typeCheckingMode = "basic",
              autoSearchPaths = true,
              useLibraryCodeForTypes = true,
            },
          },
        },
      },
    },
    ruff_lsp = {
      init_options = {
        settings = {
          args = {},
        },
      },
    },
  },
},
},

```

```
}
```

Web Development

```
-- ~/.config/nvim/lua/plugins/web.lua

return {
  -- TypeScript/JavaScript
  {
    "neovim/nvim-lspconfig",
    opts = {
      servers = {
        tsserver = {
          settings = {
            typescript = {
              inlayHints = {
                includeInlayParameterNameHints = "literal",
                includeInlayParameterNameHintsWhenArgumentMatchesName = false,
                includeInlayFunctionParameterTypeHints = true,
                includeInlayVariableTypeHints = false,
                includeInlayPropertyDeclarationTypeHints = true,
                includeInlayFunctionLikeReturnTypeHints = true,
                includeInlayEnumMemberValueHints = true,
              },
            },
          },
          javascript = {
            inlayHints = {
              includeInlayParameterNameHints = "all",
              includeInlayParameterNameHintsWhenArgumentMatchesName = false,
              includeInlayFunctionParameterTypeHints = true,
              includeInlayVariableTypeHints = true,
              includeInlayPropertyDeclarationTypeHints = true,
              includeInlayFunctionLikeReturnTypeHints = true,
              includeInlayEnumMemberValueHints = true,
            },
          },
        },
      },
      eslint = {},
      html = {},
      cssls = {},
      tailwindcss = {},
    },
  },
}
```

```

-- Emmet for HTML/CSS
{
  "mattn/emmet-vim",
  ft = { "html", "css", "javascript", "typescript", "javascriptreact", "typescriptreact" }
},

-- Auto close tags
{
  "windwp/nvim-ts-autotag",
  ft = { "html", "javascript", "typescript", "javascriptreact", "typescriptreact", "svelte" },
  config = function()
    require("nvim-ts-autotag").setup()
  end,
},

-- Package.json management
{
  "vuki656/package-info.nvim",
  dependencies = "MunifTanjim/nui.nvim",
  ft = "json",
  config = function()
    require("package-info").setup()
  end,
},

-- REST client
{
  "rest-nvim/rest.nvim",
  dependencies = { "nvim-lua/plenary.nvim" },
  ft = "http",
  config = function()
    require("rest-nvim").setup({
      result_split_horizontal = false,
      result_split_in_place = false,
      skip_ssl_verification = false,
      encode_url = true,
      highlight = {
        enabled = true,
        timeout = 150,
      },
    },
    result = {
      show_url = true,
      show_curl_command = false,
      show_http_info = true,
      show_headers = true,
      formatters = {

```

```

        json = "jq",
        html = function(body)
            return vim.fn.system({ "tidy", "-i", "-q", "-" }, body)
        end,
    },
},
})
end,
},
}

```

Go Development

```

-- ~/.config/nvim/lua/plugins/go.lua

return {
    -- Go plugin
    {
        "ray-x/go.nvim",
        dependencies = { -- optional packages
            "ray-x/guihua.lua",
            "neovim/nvim-lspconfig",
            "nvim-treesitter/nvim-treesitter",
        },
        config = function()
            require("go").setup()
        end,
        event = { "CmdlineEnter" },
        ft = { "go", "gomod" },
        build = ':lua require("go.install").update_all_sync()', -- if you need to install/update
    },

    -- Go testing
    {
        "nvim-neotest/neotest",
        optional = true,
        dependencies = {
            "nvim-neotest/neotest-go",
        },
        opts = {
            adapters = {
                ["neotest-go"] = {
                    -- Here you can specify the settings for the adapter, i.e.
                    args = { "-count=1", "-timeout=60s" },
                },
            },
        },
    },
}

```



```

    },
  },
},

-- Go LSP configuration
{
  "neovim/nvim-lspconfig",
  opts = {
    servers = {
      gopls = {
        settings = {
          gopls = {
            gofumpt = true,
            codelenses = {
              gc_details = false,
              generate = true,
              regenerate_cgo = true,
              run_govulncheck = true,
              test = true,
              tidy = true,
              upgrade_dependency = true,
              vendor = true,
            },
            hints = {
              assignVariableTypes = true,
              compositeLiteralFields = true,
              compositeLiteralTypes = true,
              constantValues = true,
              functionTypeParameters = true,
              parameterNames = true,
              rangeVariableTypes = true,
            },
            analyses = {
              fieldalignment = true,
              nilness = true,
              unusedparams = true,
              unusedwrite = true,
              useany = true,
            },
            usePlaceholders = true,
            completeUnimported = true,
            staticcheck = true,
            directoryFilters = { "-.git", "-.vscode", "-.idea", "-.vscode-test", "-node_modules"},
            semanticTokens = true,
          },
        },
      },
    },
  },
},

```

```

    },
  },
},

-- Go debugging
{
  "leoluz/nvim-dap-go",
  ft = "go",
  dependencies = "mfussenegger/nvim-dap",
  config = function(_, opts)
    require("dap-go").setup(opts)
  end,
},
}

```

Customization Examples

Custom Colorscheme

```

-- ~/.config/nvim/lua/plugins/colorscheme.lua

return {
  -- Catppuccin theme
  {
    "catppuccin/nvim",
    name = "catppuccin",
    priority = 1000,
    opts = {
      flavour = "mocha", -- latte, frappe, macchiato, mocha
      background = { -- :h background
        light = "latte",
        dark = "mocha",
      },
      transparent_background = false,
      show_end_of_buffer = false,
      term_colors = false,
      dim_inactive = {
        enabled = false,
        shade = "dark",
        percentage = 0.15,
      },
      no_italic = false,
      no_bold = false,
      no_underline = false,
      styles = {

```

```

        comments = { "italic" },
        conditionals = { "italic" },
        loops = {},
        functions = {},
        keywords = {},
        strings = {},
        variables = {},
        numbers = {},
        booleans = {},
        properties = {},
        types = {},
        operators = {},
    },
    color_overrides = {},
    custom_highlights = {},
    integrations = {
        cmp = true,
        gitsigns = true,
        nvimtree = true,
        telescope = true,
        notify = false,
        mini = false,
    },
},
},
},

-- Configure LazyVim to load catppuccin
{
    "LazyVim/LazyVim",
    opts = {
        colorscheme = "catppuccin",
    },
},
}
}

```

Custom Dashboard

```

-- ~/.config/nvim/lua/plugins/dashboard.lua

return {
    {
        "nvimdev/dashboard-nvim",
        event = "VimEnter",
        opts = function()
            local logo = [[

```

```

                                Z
                                Z
                                z
                                z
                                z

]]

logo = string.rep("\n", 8) .. logo .. "\n\n"

local opts = {
  theme = "doom",
  hide = {
    statusline = false,
  },
  config = {
    header = vim.split(logo, "\n"),
    center = {
      { action = "Telescope find_files", desc = " Find file", icon = " ", key = "f" },
      { action = "ene | startinsert", desc = " New file", icon = " ", key = "n" },
      { action = "Telescope oldfiles", desc = " Recent files", icon = " ", key = "r" },
      { action = "Telescope live_grep", desc = " Find text", icon = " ", key = "g" },
      { action = [[lua require("lazyvim.util").telescope.config_files()()]], desc = " "},
      { action = 'lua require("persistence").load()', desc = " Restore Session", icon = " "},
      { action = "LazyExtras", desc = " Lazy Extras", icon = " ", key = "x" },
      { action = "Lazy", desc = " Lazy", icon = " ", key = "l" },
      { action = "qa", desc = " Quit", icon = " ", key = "q" },
    },
    footer = function()
      local stats = require("lazy").stats()
      local ms = (math.floor(stats.startuptime * 100 + 0.5) / 100)
      return { " Neovim loaded " .. stats.loaded .. "/" .. stats.count .. " plugins in " .. ms .. "ms" }
    end,
  },
}

for _, button in ipairs(opts.config.center) do
  button.desc = button.desc .. string.rep(" ", 43 - #button.desc)
  button.key_format = " %s"
end

-- close Lazy and re-open when the dashboard is ready
if vim.o.filetype == "lazy" then
  vim.cmd.close()
  vim.api.nvim_create_autocmd("User", {
    pattern = "DashboardLoaded",
    callback = function()

```

```

        require("lazy").show()
    end,
})
end

    return opts
end,
},
}

```

Performance Optimization

Startup Optimization

```

-- ~/.config/nvim/lua/config/lazy.lua

-- Bootstrap lazy.nvim
local lazypath = vim.fn.stdpath("data") .. "/lazy/lazy.nvim"
if not vim.loop.fs_stat(lazypath) then
    vim.fn.system({
        "git",
        "clone",
        "--filter=blob:none",
        "https://github.com/folke/lazy.nvim.git",
        "--branch=stable",
        lazypath,
    })
end
vim.opt.rtp:prepend(lazypath)

require("lazy").setup({
    spec = {
        -- add LazyVim and import its plugins
        { "LazyVim/LazyVim", import = "lazyvim.plugins" },
        -- import any extras modules here
        { import = "lazyvim.plugins.extras.lang.typescript" },
        { import = "lazyvim.plugins.extras.lang.json" },
        { import = "lazyvim.plugins.extras.ui.mini-animate" },
        -- import/override with your plugins
        { import = "plugins" },
    },
    defaults = {
        lazy = false, -- should plugins be lazy-loaded?
        version = false, -- always use the latest git commit
    }
})

```

```

},
install = { colorscheme = { "tokyonight", "habamax" } },
checker = { enabled = true }, -- automatically check for plugin updates
performance = {
  rtp = {
    -- disable some rtp plugins
    disabled_plugins = {
      "gzip",
      "matchit",
      "matchparen",
      "netrwPlugin",
      "tarPlugin",
      "tohtml",
      "tutor",
      "zipPlugin",
    },
  },
},
},
})

```

Memory Optimization

```

-- ~/.config/nvim/lua/plugins/performance.lua

return {
  -- Optimize treesitter
  {
    "nvim-treesitter/nvim-treesitter",
    opts = {
      highlight = { enable = true },
      indent = { enable = true },
      ensure_installed = {
        "bash",
        "c",
        "html",
        "javascript",
        "json",
        "lua",
        "luadoc",
        "luap",
        "markdown",
        "markdown_inline",
        "python",
        "query",
        "regex",

```

```

        "tsx",
        "typescript",
        "vim",
        "vimdoc",
        "yaml",
    },
    incremental_selection = {
        enable = true,
        keymaps = {
            init_selection = "<C-space>",
            node_incremental = "<C-space>",
            scope_incremental = false,
            node_decremental = "<bs>",
        },
    },
},
},
},

-- Optimize LSP
{
    "neovim/nvim-lspconfig",
    opts = {
        diagnostics = {
            underline = true,
            update_in_insert = false,
            virtual_text = {
                spacing = 4,
                source = "if_many",
                prefix = " ",
            },
            severity_sort = true,
        },
        inlay_hints = {
            enabled = true,
        },
        capabilities = {},
        format = {
            formatting_options = nil,
            timeout_ms = nil,
        },
        servers = {},
        setup = {},
    },
},
}

```

Troubleshooting

Common Issues

```
# Check LazyVim health
:checkhealth

# Check specific components
:checkhealth lazy
:checkhealth mason
:checkhealth telescope
:checkhealth treesitter

# Plugin management
:Lazy          # Open Lazy plugin manager
:Lazy sync     # Update all plugins
:Lazy clean    # Remove unused plugins
:Lazy profile  # Profile startup time

# LSP troubleshooting
:LspInfo       # Show LSP information
:LspRestart    # Restart LSP servers
:Mason         # Open Mason (LSP installer)

# Clear cache
:lua vim.fn.delete(vim.fn.stdpath("cache"), "rf")
```

Reset Configuration

```
# Backup current config
mv ~/.config/nvim ~/.config/nvim.backup
mv ~/.local/share/nvim ~/.local/share/nvim.backup
mv ~/.local/state/nvim ~/.local/state/nvim.backup
mv ~/.cache/nvim ~/.cache/nvim.backup

# Fresh install
git clone https://github.com/LazyVim/starter ~/.config/nvim
rm -rf ~/.config/nvim/.git
```


Performance Issues

```
-- Profile startup time
-- Add to ~/.config/nvim/lua/config/options.lua
vim.g.loaded_python3_provider = 0
vim.g.loaded_ruby_provider = 0
vim.g.loaded_perl_provider = 0
vim.g.loaded_node_provider = 0

-- Disable unused features
vim.g.loaded_gzip = 1
vim.g.loaded_zip = 1
vim.g.loaded_zipPlugin = 1
vim.g.loaded_tar = 1
vim.g.loaded_tarPlugin = 1
vim.g.loaded_getscript = 1
vim.g.loaded_getscriptPlugin = 1
vim.g.loaded_vimball = 1
vim.g.loaded_vimballPlugin = 1
vim.g.loaded_2html_plugin = 1
vim.g.loaded_logiPat = 1
vim.g.loaded_rrhelper = 1
vim.g.loaded_netrw = 1
vim.g.loaded_netrwPlugin = 1
vim.g.loaded_netrwSettings = 1
vim.g.loaded_netrwFileHandlers = 1
```

LazyVim provides an excellent foundation for a modern Neovim setup, combining the power of Neovim with sensible defaults and a beautiful interface. It's perfect for developers who want a powerful IDE-like experience without the complexity of building a configuration from scratch.

Helix

Helix: A Modern Modal Editor

Helix is a modern, terminal-based text editor written in Rust. It takes inspiration from Kakoune and Vim, featuring multiple selections, built-in LSP support, and a focus on simplicity and performance.

What is Helix?

Helix is designed with modern editing principles: - **Selection-first editing**: Select text first, then act on it - **Multiple selections**: Edit multiple locations simultaneously - **Built-in LSP**: Language Server Protocol support out of the box - **Tree-sitter**: Advanced syntax highlighting and text objects - **No configuration required**: Works great with zero configuration - **Fast and lightweight**: Written in Rust for performance

Why Choose Helix?

- **Modern design**: Built from the ground up with modern editing concepts
- **Zero configuration**: Sensible defaults that work immediately
- **Multiple selections**: Powerful multi-cursor editing
- **Built-in features**: LSP, tree-sitter, and fuzzy finder included
- **Consistent keybindings**: Logical and memorable key combinations
- **Linux-optimized**: Designed with Linux development workflows in mind
- **Active development**: Regular updates and improvements
- **Rust performance**: Fast and memory-efficient on Linux systems

Installation

Package Managers

```
# Ubuntu/Debian (from source)
sudo apt update
sudo apt install git rustc cargo
git clone https://github.com/helix-editor/helix
cd helix
cargo install --path helix-term --locked

# Arch Linux
```

```
sudo pacman -S helix

# Fedora
sudo dnf install helix

# SUSE/openSUSE
sudo zypper install helix

# Gentoo
sudo emerge app-editors/helix

# Void Linux
sudo xbps-install helix

# Nix
nix-env -iA nixpkgs.helix

# Cargo (from source)
cargo install helix-term --locked
```

Building from Source

```
# Clone repository
git clone https://github.com/helix-editor/helix
cd helix

# Build and install
cargo install --path helix-term --locked

# Install runtime files
mkdir -p ~/.config/helix
cp -r runtime ~/.config/helix/

# Or use system-wide installation
sudo mkdir -p /usr/local/lib/helix
sudo cp -r runtime /usr/local/lib/helix/
```

Post-Installation Setup

```
# Install language servers (optional but recommended)
# JavaScript/TypeScript
npm install -g typescript-language-server typescript
```

```
# Python
pip install python-lsp-server

# Rust (if not already installed)
rustup component add rust-analyzer

# Go
go install golang.org/x/tools/gopls@latest

# Check health and see what's available
hx --health
```

Basic Concepts

Selection-First Editing

Unlike Vim's action-first approach, Helix uses selection-first:

```
# Vim way: action + motion
dw    # Delete word (action first)

# Helix way: selection + action
w     # Select word
d     # Delete selection
```

Multiple Selections

Helix excels at multiple selections:

```
# Select all occurrences of current selection
%     # Select all matches

# Add selection
C     # Duplicate cursor/selection
Alt-C # Remove primary selection

# Split selections
S     # Split selection on regex
Alt-s # Split selection on lines
```

Modes

Helix has several modes: - **Normal mode**: Default mode for navigation and commands - **Insert mode**: For typing text - **Select mode**: For making selections - **Command mode**: For ex-style

commands

Essential Commands

Starting Helix

```
# Open file
hx filename.txt

# Open multiple files
hx file1.txt file2.txt

# Open directory (file picker)
hx .

# Open with specific line/column
hx filename.txt:25:10

# Health check
hx --health
```

Basic Navigation

```
# Character movement
h      # Left
j      # Down
k      # Up
l      # Right

# Word movement
w      # Next word start
b      # Previous word start
e      # Next word end

# Line movement
O      # Line start
^      # First non-whitespace
$      # Line end
g h    # Line start (alternative)
g l    # Line end (alternative)

# Document movement
g g    # Document start
g e    # Document end
```

```

Ctrl+u # Page up
Ctrl+d # Page down

# Paragraph movement
[      # Previous paragraph
]      # Next paragraph

```

Text Selection

```

# Basic selection
v      # Enter select mode
V      # Select line
Ctrl+v # Select block (not implemented yet)

# Word/object selection
w      # Select word
W      # Select WORD (whitespace-separated)
p      # Select paragraph
f<char> # Select to character
t<char> # Select until character

# Extend selection
;      # Flip selection direction
Alt+;  # Ensure forward selection direction
%      # Select matching bracket
m      # Select matching pair (quotes, brackets, etc.)

# Multiple selections
C      # Duplicate cursor
Alt+C  # Remove primary cursor
s      # Select regex matches within selection
S      # Split selection on regex
Alt-s  # Split selection on newlines
&      # Align selections
-      # Trim whitespace from selections

```

Editing Commands

```

# Insert modes
i      # Insert before selection
a      # Insert after selection
I      # Insert at line start
A      # Insert at line end
o      # Insert new line below

```

```

O      # Insert new line above

# Delete/Change
d      # Delete selection
c      # Change selection (delete and enter insert mode)
Alt+d  # Delete line
Alt+c  # Change line

# Copy/Paste
y      # Yank (copy) selection
p      # Paste after selection
P      # Paste before selection
Alt+p  # Paste every yanked selection
R      # Replace selection with yanked text

# Undo/Redo
u      # Undo
U      # Redo
Alt+u  # Earlier in time
Alt+U  # Later in time

```

Search and Replace

```

# Search
/      # Search forward
?      # Search backward
n      # Next search result
N      # Previous search result
*      # Search word under cursor

# Replace
r<char> # Replace character
~      # Switch case
`      # Set case to lowercase
Alt+`  # Set case to uppercase

# Global replace (in command mode)
:s/old/new/g      # Replace in current selection
:%s/old/new/g     # Replace in entire file
:s/old/new/gc     # Replace with confirmation

```


File and Buffer Management

File Operations

```
# File commands (in command mode)
:o filename    # Open file
:w            # Write file
:w filename    # Write as filename
:q            # Quit
:q!           # Quit without saving
:wq           # Write and quit

# Buffer navigation
Space b       # Buffer picker
gn            # Next buffer
gp            # Previous buffer
Space x       # Close buffer
```

File Picker and Fuzzy Finding

```
# File operations
Space f       # File picker
Space F       # File picker (from cwd)
Space b       # Buffer picker
Space s       # Symbol picker (LSP)
Space S       # Workspace symbol picker
Space g       # Global search (grep)
Space /       # Global search in current file
Space k       # Show documentation
Space r       # Rename symbol (LSP)
Space a       # Code action (LSP)
```

Configuration

Configuration File

Helix uses TOML for configuration:

```
# ~/.config/helix/config.toml

theme = "onedark"

[editor]
```

```

line-number = "relative"
mouse = true
middle-click-paste = true
scroll-lines = 3
shell = ["sh", "-c"]
text-width = 80
completion-trigger-len = 2
auto-completion = true
auto-format = true
auto-save = false
idle-timeout = 400
preview-completion-insert = true
completion-replace = false
auto-info = true
true-color = false
rulers = [80, 120]
bufferline = "always"
color-modes = false

[editor.statusline]
left = ["mode", "spinner"]
center = ["file-name"]
right = ["diagnostics", "selections", "position", "file-encoding", "file-line-ending", "file-encoding"]
separator = " "
mode.normal = "NORMAL"
mode.insert = "INSERT"
mode.select = "SELECT"

[editor.lsp]
display-messages = true
auto-signature-help = true
display-inlay-hints = true
display-signature-help-docs = true
snippets = true
goto-reference-include-declaration = true

[editor.cursor-shape]
insert = "bar"
normal = "block"
select = "underline"

[editor.file-picker]
hidden = true
follow-symlinks = true
deduplicate-links = true
parents = true
ignore = true

```

```

git-ignore = true
git-global = true
git-exclude = true
max-depth = 25

[editor.auto-pairs]
'(' = ')'
'{' = '}'
'[' = ']'
'"' = '"'
"'" = "'"
"`" = "`"
'<' = '>'

[editor.search]
smart-case = true
wrap-around = true

[editor.whitespace]
render = "all"
characters = { space = ".", nbsp = " ", tab = "→", newline = "␣", tabpad = "." }

[editor.indent-guides]
render = true
character = " "
skip-levels = 1

[editor.gutters]
layout = ["diff", "diagnostics", "line-numbers", "spacer"]

[editor.soft-wrap]
enable = false
max-wrap = 25
max-indent-retain = 0
wrap-indicator = ""

[keys.normal]
# Custom keybindings
C-s = ":w"
C-o = "file_picker"
C-p = "file_picker"
esc = ["collapse_selection", "keep_primary_selection"]

[keys.insert]
# Insert mode keybindings
C-s = ["normal_mode", ":w", "insert_mode"]
j = { k = "normal_mode" } # jk to exit insert mode

```

```
[keys.select]
# Select mode keybindings
esc = ["collapse_selection", "keep_primary_selection", "normal_mode"]
```

Language Configuration

```
# ~/.config/helix/languages.toml

# Override language server configurations
[[language]]
name = "python"
language-server = { command = "pylsp" }
auto-format = true

[language.config.pylsp.plugins]
autopep8 = { enabled = false }
flake8 = { enabled = true }
mccabe = { enabled = false }
pycodestyle = { enabled = false }
pyflakes = { enabled = false }
pylint = { enabled = true }
yapf = { enabled = true }

[[language]]
name = "javascript"
language-server = { command = "typescript-language-server", args = ["--stdio"] }
auto-format = true

[[language]]
name = "typescript"
language-server = { command = "typescript-language-server", args = ["--stdio"] }
auto-format = true

[[language]]
name = "rust"
language-server = { command = "rust-analyzer" }
auto-format = true

[[language]]
name = "go"
language-server = { command = "gopls" }
auto-format = true

[[language]]
name = "html"
```

```

language-server = { command = "vscode-html-language-server", args = ["--stdio"] }
auto-format = true

[[language]]
name = "css"
language-server = { command = "vscode-css-language-server", args = ["--stdio"] }
auto-format = true

[[language]]
name = "json"
language-server = { command = "vscode-json-language-server", args = ["--stdio"] }
auto-format = true

# Custom language definitions
[[language]]
name = "log"
scope = "text.log"
file-types = ["log"]
comment-token = "#"
indent = { tab-width = 2, unit = "  " }

[[grammar]]
name = "log"
source = { git = "https://github.com/Tudyx/tree-sitter-log", rev = "main" }

```

Themes

Helix comes with many built-in themes:

```

# List available themes
:theme

# Change theme temporarily
:theme dracula

# Popular themes
:theme onedark
:theme gruvbox
:theme catppuccin_mocha
:theme tokyo-night
:theme solarized_dark
:theme nord

```

Custom Theme

```
# ~/.config/helix/themes/mytheme.toml

"ui.background" = { bg = "#1e1e1e" }
"ui.text" = "#d4d4d4"
"ui.text.focus" = "#ffffff"
"ui.selection" = { bg = "#264f78" }
"ui.cursor" = { fg = "#1e1e1e", bg = "#d4d4d4" }
"ui.cursor.match" = { bg = "#3a3d41" }
"ui.linenr" = "#858585"
"ui.linenr.selected" = "#c6c6c6"
"ui.statusline" = { fg = "#d4d4d4", bg = "#007acc" }
"ui.statusline.inactive" = { fg = "#d4d4d4", bg = "#3c3c3c" }
"ui.popup" = { bg = "#252526" }
"ui.window" = "#3c3c3c"
"ui.help" = { bg = "#252526" }

"error" = "#f44747"
"warning" = "#ff8c00"
"info" = "#0080ff"
"hint" = "#969696"

"diagnostic.error" = { underline = { color = "#f44747", style = "curl" } }
"diagnostic.warning" = { underline = { color = "#ff8c00", style = "curl" } }
"diagnostic.info" = { underline = { color = "#0080ff", style = "curl" } }
"diagnostic.hint" = { underline = { color = "#969696", style = "curl" } }

"syntax.comment" = "#6a9955"
"syntax.keyword" = "#569cd6"
"syntax.string" = "#ce9178"
"syntax.number" = "#b5cea8"
"syntax.boolean" = "#569cd6"
"syntax.type" = "#4ec9b0"
"syntax.function" = "#dcdcaa"
"syntax.variable" = "#9cdcfе"
"syntax.constant" = "#4fc1ff"
"syntax.operator" = "#d4d4d4"
"syntax.punctuation" = "#d4d4d4"
```

Real-World Usage Scenarios

Scenario 1: Python Development

```
# Open Python project
hx main.py

# Navigate to function definition
gd    # Go to definition (LSP)
gr    # Go to references (LSP)

# Multiple cursor editing
# Select function name
w      # Select word
%      # Select all occurrences
c      # Change all occurrences
new_function_name<Esc>

# Code formatting
:format    # Format current file
# Or enable auto-format in config

# Symbol search
Space s    # Search symbols in file
Space S    # Search symbols in workspace

# Diagnostics
]d         # Next diagnostic
[d         # Previous diagnostic
Space k    # Show hover information
Space a    # Code actions
```

Scenario 2: Web Development

```
# Open HTML file
hx index.html

# Select HTML tag
mt      # Select matching tag pair
c       # Change tag
div<Esc>

# Multiple selections for attributes
# Select class attribute value
f"      # Find quote
```

```

mi      # Select inside quotes
%       # Select all similar
c       # Change all
new-class<Esc>

# Navigate between files
Space f  # File picker
# Type filename and Enter

# Global search and replace
Space g  # Global search
# Type search term
# Navigate results with j/k
# Press Enter to go to location

```

Scenario 3: Configuration File Editing

```

# Open config file
hx ~/.config/helix/config.toml

# Navigate to specific section
/editor  # Search for "editor"
n        # Next occurrence

# Select and modify values
f=       # Find equals sign
l        # Move right
w        # Select value
c        # Change value
new_value<Esc>

# Comment/uncomment lines
Alt+;    # Toggle line comment
# Or select multiple lines and comment
V        # Select line
jjj      # Extend selection
Alt+;    # Comment selected lines

```

Scenario 4: Log File Analysis

```

# Open log file
hx /var/log/application.log

# Search for errors

```



```

/ERROR      # Search for ERROR
n           # Next occurrence
N           # Previous occurrence

# Select all error lines
*           # Search word under cursor (ERROR)
%           # Select all matches
y           # Yank all error lines

# Open new buffer for analysis
:new
p           # Paste error lines

# Split selections by timestamp
s^\d{4}-\d{2}-\d{2} # Split on date pattern

```

Advanced Features

Multiple Selections Mastery

```

# Complex selection scenarios
# Select all function calls
/\w+\(      # Search for function calls
%           # Select all matches

# Select inside parentheses for all matches
mi          # Select inside matching pairs

# Add numbers to each selection
C           # Duplicate cursor to each selection
I           # Insert at beginning
1. <Esc>    # Add numbering (manual)

# Or use regex replacement
:s/^/1. /   # Add numbering with regex

```

Text Objects

```

# Built-in text objects
mw          # Select word
mp          # Select paragraph
m(          # Select inside parentheses
m)          # Select around parentheses

```

```

m[    # Select inside brackets
m]    # Select around brackets
m{    # Select inside braces
m}    # Select around braces
m"    # Select inside quotes
m'    # Select inside single quotes
m`    # Select inside backticks
mt    # Select inside HTML/XML tags
mf    # Select function
mc    # Select class
ma    # Select argument

```

Tree-sitter Navigation

```

# Navigate syntax tree
Alt+p # Parent node
Alt+c # Child node
Alt+n # Next sibling
Alt+o # Previous sibling

# Select syntax nodes
mf    # Select function
mc    # Select class
ma    # Select argument/parameter

```

LSP Features

```

# Code navigation
gd    # Go to definition
gy    # Go to type definition
gr    # Go to references
gi    # Go to implementation

# Code information
Space k # Hover information
Space K # Signature help

# Code modification
Space r # Rename symbol
Space a # Code actions
:format # Format document

# Diagnostics
]d      # Next diagnostic

```

```
[d      # Previous diagnostic
]D      # Next error
[D      # Previous error
Space d # Show diagnostics
```

Customization and Scripting

Custom Commands

```
# ~/.config/helix/config.toml

[keys.normal]
# Custom command sequences
C-g = [":new", ":insert-output echo 'Hello World'"]
C-t = [":sh date"]

# Macro-like sequences
F1 = ["select_all", "yank", ":new", "paste_after"]
F2 = ["goto_line_start", "insert_mode", "// ", "normal_mode"]
```

Integration with External Tools

```
# Pipe selection to external command
|sort      # Sort selected lines
|uniq      # Remove duplicate lines
|wc -l     # Count lines
|jq        # Format JSON
|xmllint --format - # Format XML

# Insert command output
:insert-output date
:insert-output ls -la
:insert-output git log --oneline -10
```

Shell Integration

```
# Set Helix as default editor
export EDITOR=hx
export VISUAL=hx

# Git integration
```

```
git config --global core.editor hx

# Shell aliases
alias h='hx'
alias hx.='hx .'
alias hxc='hx ~/.config/helix/config.toml'
```

Performance and Optimization

Large File Handling

Helix handles large files efficiently:

```
# Open large files
hx large_file.log

# Navigate efficiently
gg      # Go to start
ge      # Go to end
Ctrl+u  # Page up
Ctrl+d  # Page down

# Search in large files
/pattern # Incremental search
n        # Next match
N        # Previous match
```

Memory Usage

```
# Check memory usage
:debug-info

# Optimize for memory
# Disable unnecessary features in config
[editor]
auto-completion = false
auto-info = false
```

Troubleshooting

Common Issues

```
# Check health
hx --health

# Language server issues
:lsp-workspace-command
:lsp-restart

# Theme issues
:theme default
:reload-config

# Key binding conflicts
# Check config.toml for conflicting bindings
```

Debug Mode

```
# Start with debug logging
RUST_LOG=debug hx

# Check logs
tail -f ~/.cache/helix/helix.log
```

Reset Configuration

```
# Backup current config
mv ~/.config/helix ~/.config/helix.backup

# Start fresh
hx # Will create default config
```

Migration from Other Editors

From Vim/Neovim

Key differences to remember: - Selection-first editing - Different keybindings for some operations - Built-in LSP and tree-sitter - No plugin system (yet)

From VSCode

Helix provides many VSCode-like features: - Built-in LSP support - Fuzzy file finding - Multiple selections - Integrated terminal (external)

From Kakoune

Helix is inspired by Kakoune: - Similar selection-first philosophy - Multiple selections - Different keybindings and features

Helix offers a modern, efficient editing experience with powerful features built-in. Its selection-first approach and multiple selections make it particularly effective for complex text manipulation tasks, while the zero-configuration philosophy means you can be productive immediately. ## Next steps

- [tmux](#) for splits and SSH detach
- [Herdr](#) when running AI agent CLIs beside Helix
- [Modern Tools](#) — rg, fd, fzf, bat
- [Workflows](#)

Micro

Micro: A Modern Terminal Editor

Micro is a modern, intuitive terminal-based text editor that aims to be easy to use and easy to remember. It provides a familiar interface similar to GUI editors while running in the terminal, making it perfect for users transitioning from graphical editors.

What is Micro?

Micro is designed with simplicity and usability in mind: - **Intuitive keybindings**: Familiar Ctrl+S, Ctrl+C, Ctrl+V shortcuts - **Mouse support**: Click to position cursor, select text, scroll - **Syntax highlighting**: Built-in support for 100+ languages - **Plugin system**: Extensible with Lua plugins - **Multiple cursors**: Edit multiple locations simultaneously - **Linux-optimized**: Excellent performance on Linux systems - **No configuration required**: Works great out of the box

Why Choose Micro?

- **Easy to learn**: Familiar keybindings from GUI editors
- **Modern features**: Multiple cursors, syntax highlighting, plugins
- **Lightweight**: Single binary with no dependencies, perfect for Linux servers
- **Customizable**: Extensive configuration options
- **Active development**: Regular updates and improvements
- **Terminal-based**: Works over SSH and in minimal Linux environments
- **Unicode support**: Full Unicode and emoji support
- **System administration**: Great for Linux configuration file editing

Installation

Package Managers

```
# Ubuntu/Debian
sudo apt update
sudo apt install micro

# Fedora
sudo dnf install micro

# Arch Linux
```



```
sudo pacman -S micro

# SUSE/openSUSE
sudo zypper install micro

# Gentoo
sudo emerge app-editors/micro

# Void Linux
sudo xbps-install micro

# Snap (Universal Linux)
sudo snap install micro --classic

# Flatpak
flatpak install flathub io.github.micro-editor.micro
```

Direct Download

```
# Download latest release
curl https://getmic.ro | bash

# Or manually download
wget https://github.com/zyedidia/micro/releases/download/v2.0.11/micro-2.0.11-linux64.tar.gz
tar -xzf micro-2.0.11-linux64.tar.gz
sudo mv micro-2.0.11/micro /usr/local/bin/

# Make executable
chmod +x /usr/local/bin/micro
```

From Source

```
# Install Go if not already installed
# Then build micro
git clone https://github.com/zyedidia/micro
cd micro
make build
sudo mv micro /usr/local/bin/
```

Basic Usage

Starting Micro

```
# Open new file
micro

# Open existing file
micro filename.txt

# Open multiple files
micro file1.txt file2.txt

# Open directory (file browser)
micro .

# Open with specific settings
micro -colorscheme monokai filename.txt
micro -syntax python script.py
```

Interface Overview

```
filename.txt
This is the content of your file.
You can type directly here.
Mouse clicks work for positioning cursor.
```

Ctrl+Q Quit Ctrl+S Save Ctrl+O Open Ctrl+F Find Ctrl+Z Undo Ctrl+Y Redo

Essential Keybindings

File Operations

```
# File management
Ctrl+O      # Open file
Ctrl+S      # Save file
Ctrl+W      # Close current buffer
Ctrl+Q      # Quit micro
Ctrl+E      # Command mode
Ctrl+G      # Open help

# Buffer navigation
Ctrl+Tab    # Next buffer
Ctrl+Shift+Tab # Previous buffer
Alt+,,      # Previous buffer
Alt+.,      # Next buffer
```

Navigation

```
# Basic movement
Arrow keys  # Move cursor
Ctrl+Left/Right # Move by word
Ctrl+Up/Down # Move by paragraph
Home/End    # Beginning/end of line
Ctrl+Home/End # Beginning/end of file
Page Up/Down # Page navigation

# Advanced navigation
Ctrl+G      # Go to line
Ctrl+F      # Find
Ctrl+N      # Find next
Ctrl+P      # Find previous
```

Text Editing

```
# Basic editing
Ctrl+Z      # Undo
Ctrl+Y      # Redo
Ctrl+C      # Copy
Ctrl+X      # Cut
Ctrl+V      # Paste
Ctrl+A      # Select all
```

```
Ctrl+D          # Duplicate line
Ctrl+K          # Delete line

# Advanced editing
Ctrl+L          # Select line
Ctrl+Shift+Up   # Move line up
Ctrl+Shift+Down # Move line down
Alt+Shift+Left  # Select word left
Alt+Shift+Right # Select word right
```

Multiple Cursors

```
# Multiple cursor operations
Ctrl+MouseClick # Add cursor at click position
Alt+N           # Create cursor at next word occurrence
Alt+P           # Create cursor at previous word occurrence
Alt+C           # Create cursor at all word occurrences
Alt+M           # Remove all cursors except primary
Escape          # Remove all extra cursors
```

Search and Replace

```
# Search operations
Ctrl+F          # Find
Ctrl+N          # Find next
Ctrl+P          # Find previous
Ctrl+R          # Find and replace

# In find/replace dialog
Tab             # Switch between find and replace fields
Enter           # Execute search/replace
Escape          # Cancel dialog
```

Configuration

Settings File

Micro stores settings in `~/.config/micro/settings.json`:

```
{
  "autoclose": true,
  "autoindent": true,
```

```

"autosave": 0,
"autosu": false,
"backup": true,
"backupdir": "",
"basename": false,
"colorcolumn": 0,
"colorscheme": "default",
"cursorline": true,
"diffgutter": false,
"divchars": "|-",
"divreverse": true,
"encoding": "utf-8",
"eofnewline": true,
"fastdirty": false,
"fileformat": "unix",
"filetype": "unknown",
"hlsearch": false,
"ignorecase": false,
"indentchar": " ",
"infobar": true,
"keepautoindent": false,
"keymenu": false,
"matchbrace": true,
"matchbraceleft": false,
"mouse": true,
"parsecursor": false,
"paste": false,
"pluginchannels": [
    "https://raw.githubusercontent.com/micro-editor/plugin-channel/master/channel.json"
],
"pluginrepos": [],
"readonly": false,
"rmtrailingws": false,
"ruler": true,
"savecursor": false,
"savehistory": true,
"saveundo": false,
"scrollbar": false,
"scrollmargin": 3,
"scrollspeed": 2,
"smartpaste": true,
"softwrap": false,
"splitbottom": true,
"splitright": true,
"statusformatl": "$(filename) $(modified)$(line),$(col)) $(status.paste)| ft:$(opt:file
"statusformatr": "$(bind:ToggleKeyMenu): bindings, $(bind:ToggleHelp): help",
"statusline": true,

```

```
"sucmd": "sudo",
"syntax": true,
"tabmovement": false,
"tabsize": 4,
"tabstospaces": false,
"termtitle": false,
"useprimary": true,
"wordwrap": false,
"xterm": false
}
```

Common Configuration Changes

```
# Set tab size to 2
micro -option tabsize 2

# Enable line numbers
micro -option ruler true

# Set color scheme
micro -option colorscheme monokai

# Enable soft wrapping
micro -option softwrap true

# Auto-save every 10 seconds
micro -option autosave 10

# Remove trailing whitespace on save
micro -option rmtrailingws true

# Make changes permanent
# These settings are automatically saved to settings.json
```

Keybinding Customization

Create ~/.config/micro/bindings.json:

```
{
  "Alt+/" : "lua:comment.comment",
  "Ctrl+/" : "lua:comment.comment",
  "Ctrl+Shift+D" : "DuplicateLine",
  "Ctrl+Shift+K" : "DeleteLine",
  "Ctrl+Shift+Up" : "MoveLinesUp",
```

```

    "Ctrl+Shift+Down": "MoveLinesDown",
    "Ctrl+J": "JumpLine",
    "Ctrl+B": "ToggleBookmark",
    "F3": "FindNext",
    "Shift+F3": "FindPrevious",
    "F4": "Quit",
    "Ctrl+T": "NewTab",
    "Ctrl+Shift+T": "ReopenLastClosedTab"
}

```

Color Schemes

Micro comes with many built-in color schemes:

```

# List available color schemes
micro -option colorscheme help

# Popular color schemes
micro -option colorscheme monokai
micro -option colorscheme solarized
micro -option colorscheme dracula
micro -option colorscheme gruvbox
micro -option colorscheme atom-dark
micro -option colorscheme github
micro -option colorscheme zenburn

```

Custom Color Scheme

Create `~/.config/micro/colorschemes/mytheme.micro`:

```

color-link default "#d4d4d4,#1e1e1e"
color-link comment "#6a9955"
color-link identifier "#9cdcfe"
color-link constant "#4fc1ff"
color-link statement "#569cd6"
color-link symbol "#d4d4d4"
color-link preproc "#c586c0"
color-link type "#4ec9b0"
color-link special "#dcdcaa"
color-link underlined "#d4d4d4"
color-link error "#f44747"
color-link todo "#ffcc00"
color-link hlsearch "#264f78"
color-link statusline "#d4d4d4,#007acc"
color-link tabbar "#d4d4d4,#2d2d30"

```

```
color-link indent-char "#3c3c3c"
color-link line-number "#858585"
color-link current-line-number "#c6c6c6"
color-link diff-added "#28a745"
color-link diff-modified "#ffd33d"
color-link diff-deleted "#d73a49"
color-link gutter-error "#f44747"
color-link gutter-warning "#ff8c00"
color-link cursor-line "#2a2d2e"
color-link color-column "#3c3c3c"
```

Plugin System

Plugin Management

```
# List installed plugins
micro -plugin list

# Install plugin
micro -plugin install pluginname

# Remove plugin
micro -plugin remove pluginname

# Update plugins
micro -plugin update

# Search for plugins
micro -plugin search keyword

# Plugin information
micro -plugin info pluginname
```

Popular Plugins

# Essential plugins	
micro -plugin install filemanager	# File browser
micro -plugin install linter	# Code linting
micro -plugin install comment	# Toggle comments
micro -plugin install jump	# Quick navigation
micro -plugin install manipulator	# Text manipulation
micro -plugin install quoter	# Quote manipulation
micro -plugin install snippets	# Code snippets
micro -plugin install autoclose	# Auto-close brackets


```

micro -plugin install bookmark      # Bookmarks
micro -plugin install diff          # Diff viewer

# Language-specific plugins
micro -plugin install go            # Go language support
micro -plugin install python        # Python support
micro -plugin install html          # HTML support
micro -plugin install css           # CSS support
micro -plugin install javascript    # JavaScript support

```

Plugin Configuration

Plugins can be configured in `~/.config/micro/settings.json`:

```

{
  "linter": true,
  "linter.ignoredMessages": [],
  "comment.leader": "# ",
  "filemanager.showdotfiles": true,
  "filemanager.showignored": false,
  "snippets.enabled": true,
  "autoclose.enabled": true
}

```

Real-World Usage Scenarios

Scenario 1: Linux System Configuration

```

# Edit SSH daemon configuration
sudo micro /etc/ssh/sshd_config

# Navigate to specific setting
Ctrl+F
# Type "PasswordAuthentication"
Enter

# Change the value
# Use mouse to click and position cursor
# Change "yes" to "no"

# Save and restart SSH service
Ctrl+S
Ctrl+Q

```

```

sudo systemctl restart sshd

# Alternative: Edit systemd service file
sudo micro /etc/systemd/system/myapp.service

# Add service configuration
[Unit]
Description=My Application
After=network.target

[Service]
Type=simple
User=myapp
ExecStart=/usr/local/bin/myapp
Restart=always

[Install]
WantedBy=multi-user.target

# Save and enable service
Ctrl+S
Ctrl+Q
sudo systemctl daemon-reload
sudo systemctl enable myapp
sudo systemctl start myapp

```

Scenario 2: Python Development

```

# Open Python file
micro script.py

# Enable Python-specific settings
# (Automatically detected by file extension)

# Multiple cursor editing for variable renaming
# Double-click on variable name
Alt+C # Select all occurrences
# Type new variable name

# Comment/uncomment code blocks
# Select lines with mouse or Shift+arrows
Ctrl+/ # Toggle comments

# Duplicate lines for testing
Ctrl+D # Duplicate current line

```

```
# Save file  
Ctrl+S
```

Scenario 3: Log File Analysis

```
# Open log file  
micro /var/log/application.log  
  
# Search for errors  
Ctrl+F  
# Type "ERROR"  
Enter  
  
# Navigate through errors  
Ctrl+N # Next occurrence  
Ctrl+P # Previous occurrence  
  
# Select error lines for copying  
# Click and drag to select  
Ctrl+C # Copy  
  
# Open new buffer for analysis  
Ctrl+E  
# Type "tab new"  
Enter  
  
# Paste errors  
Ctrl+V  
  
# Save analysis  
Ctrl+S  
# Type filename  
Enter
```

Scenario 4: Web Development

```
# Open HTML file  
micro index.html  
  
# Use multiple cursors for tag editing  
# Select opening tag  
Alt+C # Select all similar tags  
# Edit all tags simultaneously
```

```
# Navigate between files
Ctrl+O
# Type filename or use arrow keys
Enter

# Split view for CSS editing
Ctrl+E
# Type "hsplit styles.css"
Enter

# Switch between splits
Ctrl+W # Cycle through splits

# Format code (if formatter plugin installed)
Ctrl+E
# Type "format"
Enter
```

Advanced Features

Command Mode

Access command mode with Ctrl+E:

```
# File operations
open filename.txt      # Open file
save                   # Save current file
save filename.txt      # Save as
quit                   # Quit current buffer
exit                   # Exit micro

# Buffer operations
tab new                # New tab
tab close              # Close current tab
tab switch N           # Switch to tab N
buffer N               # Switch to buffer N

# View operations
hsplit filename.txt    # Horizontal split
vsplit filename.txt    # Vertical split
unsplit                # Close current split

# Search and replace
find pattern           # Find pattern
replace old new        # Replace old with new
```

```
replaceall old new    # Replace all occurrences

# Settings
set option value      # Set option
show option           # Show option value
reset option          # Reset option to default

# Plugin operations
plugin install name   # Install plugin
plugin remove name    # Remove plugin
plugin list           # List plugins
plugin update         # Update plugins
```

Macros

```
# Record macro
Ctrl+U # Start recording
# Perform actions
Ctrl+U # Stop recording

# Play macro
Ctrl+J # Execute recorded macro
```

Bookmarks

```
# Set bookmark
Ctrl+B # Toggle bookmark on current line

# Navigate bookmarks
Alt+Up # Previous bookmark
Alt+Down # Next bookmark

# List bookmarks
Ctrl+E
# Type "bookmark list"
Enter
```

Splits and Tabs

```
# Split operations
Ctrl+E hsplit filename # Horizontal split
Ctrl+E vsplit filename # Vertical split
Ctrl+W                # Cycle through splits
```

```
Ctrl+Q                # Close current split

# Tab operations
Ctrl+T                # New tab
Ctrl+W                # Close tab
Ctrl+Tab              # Next tab
Ctrl+Shift+Tab        # Previous tab
```

Integration with Development Tools

Git Integration

```
# View git diff
micro -option diffgutter true filename.txt

# Git commands through shell
Ctrl+E
# Type "term git status"
Enter

# Or use external git tools
# Configure git to use micro as editor
git config --global core.editor micro
```

Language Server Integration

While micro doesn't have built-in LSP support, you can use external tools:

```
# Install language servers
npm install -g typescript-language-server
pip install python-lsp-server

# Use with external LSP client
# Or use plugins that provide LSP integration
micro -plugin install lsp
```

Build System Integration

```
# Run build commands
Ctrl+E
# Type "term make"
Enter
```

```
# Or create custom commands
# In settings.json, add:
{
  "build": "make",
  "test": "npm test",
  "run": "python main.py"
}

# Then use:
Ctrl+E
# Type "build" or "test" or "run"
Enter
```

Customization Examples

Custom Status Line

```
{
  "statusformatl": "${filename} [$(opt:filetype)] $(modified) | Line $(line)/$(lines) Col
  "statusformatr": "$(opt:encoding) | $(opt:fileformat) | Micro v$(version)"
}
```

Custom Key Bindings for Productivity

```
{
  "Ctrl+Shift+P": "lua:comment.comment",
  "Ctrl+D": "DuplicateLine",
  "Ctrl+Shift+K": "DeleteLine",
  "Ctrl+L": "SelectLine",
  "Ctrl+Shift+L": "SelectToEndOfLine",
  "F2": "Rename",
  "F5": "Reload",
  "F12": "ToggleHelp",
  "Ctrl+`": "ToggleTerminal"
}
```

File Type Specific Settings

Create ~/.config/micro/settings.json:

```

{
  "*.py": {
    "tabsize": 4,
    "tabstospaces": true,
    "rmtrailingws": true
  },
  "*.js": {
    "tabsize": 2,
    "tabstospaces": true
  },
  "*.go": {
    "tabsize": 4,
    "tabstospaces": false
  },
  "*.md": {
    "softwrap": true,
    "wordwrap": true
  }
}

```

Performance and Optimization

Large File Handling

```

# Open large files efficiently
micro -option syntax false largefile.txt

# Disable features for better performance
micro -option hlsearch false
micro -option matchbrace false
micro -option autoindent false

```

Memory Usage

```

# Check memory usage
ps aux | grep micro

# Optimize for low memory
micro -option backup false
micro -option savehistory false
micro -option saveundo false

```


Troubleshooting

Common Issues

```
# Terminal compatibility issues
export TERM=xterm-256color
micro filename.txt

# Mouse not working
micro -option mouse true filename.txt

# Colors not displaying correctly
micro -option colorscheme default filename.txt

# Plugin issues
micro -plugin remove problematic-plugin
micro -plugin update
```

Debug Mode

```
# Enable debug logging
micro -debug filename.txt

# Check log file
tail -f ~/.config/micro/log.txt
```

Reset Configuration

```
# Backup current config
mv ~/.config/micro ~/.config/micro.backup

# Start fresh (micro will create new config)
micro
```

Migration from Other Editors

From Nano

Micro provides similar simplicity with more features: - Familiar Ctrl+S, Ctrl+O shortcuts - Mouse support - Better syntax highlighting - Plugin system

From GUI Editors

Micro bridges the gap between terminal and GUI editors: - Familiar keybindings (Ctrl+C, Ctrl+V, etc.) - Mouse support - Multiple cursors - Modern interface

From Vim

For Vim users wanting something simpler: - No modal editing - Intuitive keybindings - Built-in help system - Less configuration required

Micro excels as a user-friendly terminal editor that doesn't sacrifice power for simplicity. Its familiar interface, modern features, and extensibility make it an excellent choice for developers who want a capable editor without a steep learning curve.

Kakoune

Kakoune: The Selection-Based Editor

Kakoune is a modal editor that emphasizes selection-based editing and multiple selections. Inspired by Vim but with a different philosophy, Kakoune provides powerful text manipulation capabilities through its unique approach to editing.

What is Kakoune?

Kakoune (kak) is built around several key principles: - **Selection-first editing**: Select text first, then act on it - **Multiple selections**: Edit multiple locations simultaneously - **Orthogonal design**: Commands compose naturally - **Client-server architecture**: Multiple clients can connect to one session - **Powerful text objects**: Rich set of selection primitives - **Minimal configuration**: Works well with minimal setup

Why Choose Kakoune?

- **Intuitive workflow**: Selection-first approach feels natural
- **Multiple selections**: Powerful multi-cursor editing
- **Composable commands**: Commands work together logically
- **Fast and lightweight**: Efficient C++ implementation optimized for Linux
- **Extensible**: Scriptable with shell commands and Linux utilities
- **Unique approach**: Different from Vim/Emacs paradigms
- **Active community**: Growing ecosystem of plugins and tools
- **Linux-native**: Designed with Linux development workflows in mind

Installation

Package Managers

```
# Ubuntu/Debian
sudo apt update
sudo apt install kakoune

# Fedora
sudo dnf install kakoune

# Arch Linux
```

```
sudo pacman -S kakoune

# SUSE/openSUSE
sudo zypper install kakoune

# Gentoo
sudo emerge app-editors/kakoune

# Void Linux
sudo xbps-install kakoune

# FreeBSD
pkg install kakoune

# OpenBSD
pkg_add kakoune
```

Building from Source

```
# Install dependencies
# Ubuntu/Debian
sudo apt install build-essential libncurses5-dev libncursesw5-dev

# Clone and build
git clone https://github.com/mawww/kakoune.git
cd kakoune
make
sudo make install

# Or install to custom location
make install PREFIX=$HOME/.local
```

Development Version

```
# For latest features
git clone https://github.com/mawww/kakoune.git
cd kakoune
git checkout master
make
sudo make install
```

Basic Concepts

Selection-First Philosophy

Unlike Vim's action-first approach, Kakoune uses selection-first:

```
# Vim way: action + motion
dw    # Delete word (action first)

# Kakoune way: selection + action
w     # Select word
d     # Delete selection
```

Multiple Selections

Kakoune excels at multiple selections:

```
# Select all occurrences
%      # Select all matches of current selection
Alt+%  # Select all matches in buffer

# Split selections
s      # Split selection on regex
Alt+s  # Split selection on lines
```

Modes

Kakoune has several modes: - **Normal mode**: Default mode for navigation and commands - **Insert mode**: For typing text - **Prompt mode**: For entering commands and searches - **Menu mode**: For selecting from options

Essential Commands

Starting Kakoune

```
# Open file
kak filename.txt

# Open multiple files
kak file1.txt file2.txt

# Start server session
kak -s session_name filename.txt
```

```
# Connect to existing session
kak -c session_name

# List sessions
kak -l

# Kill session
kak -c session_name -e 'kill'
```

Basic Navigation

```
# Character movement
h      # Left
j      # Down
k      # Up
l      # Right

# Word movement
w      # Next word start
b      # Previous word start
e      # Next word end
Alt+w  # Next WORD start
Alt+b  # Previous WORD start
Alt+e  # Next WORD end

# Line movement
O      # Line start
^      # First non-blank
$      # Line end
Alt+h  # Line start (alternative)
Alt+l  # Line end (alternative)

# Document movement
g g    # Document start
g e    # Document end
g k    # Buffer top
g j    # Buffer bottom

# Page movement
Ctrl+b # Page up
Ctrl+f # Page down
Ctrl+u # Half page up
Ctrl+d # Half page down
```

Text Selection

```
# Basic selection
x      # Select line
Alt+x  # Extend selection to line
s      # Select regex matches
Alt+s  # Split selection on lines

# Word selection
w      # Select word
W      # Select WORD
Alt+w  # Extend to word
Alt+W  # Extend to WORD

# Character selection
f<char> # Select to character
t<char> # Select until character
F<char> # Select to character (backward)
T<char> # Select until character (backward)

# Extend selection
Shift+<motion> # Extend selection with motion
;            # Reduce selection to cursor
Alt+;        # Flip selection direction
```

Multiple Selections

```
# Create multiple selections
C      # Copy selection to new line
Alt+C  # Copy selection to previous line
%      # Select all matches of current selection
Alt+%  # Select all matches in buffer

# Manipulate selections
Alt+k  # Keep selections matching regex
Alt+K  # Keep selections not matching regex
$      # Pipe selections through command
|      # Pipe and replace selections
!      # Insert command output

# Selection iteration
)      # Rotate main selection forward
(      # Rotate main selection backward
Alt+)  # Rotate selections forward
Alt+(  # Rotate selections backward
```


Editing Commands

```
# Insert modes
i      # Insert before selection
a      # Insert after selection
I      # Insert at line start
A      # Insert at line end
o      # Insert new line below
O      # Insert new line above

# Delete/Change
d      # Delete selection
Alt+d  # Delete to end of line
c      # Change selection (delete and insert)
Alt+c  # Change to end of line

# Copy/Paste
y      # Yank (copy) selection
p      # Paste after selection
P      # Paste before selection
Alt+p  # Paste all selections
R      # Replace selection with yanked text

# Case manipulation
~      # Switch case
`      # Convert to lowercase
Alt+`  # Convert to uppercase

# Indentation
>      # Indent
<      # Unindent
Alt+>  # Indent including empty lines
Alt+<  # Unindent including empty lines
```

Search and Replace

```
# Search
/      # Search forward
?      # Search backward
n      # Next search result
N      # Previous search result (Alt+n)
*      # Search word under cursor

# Replace
r<char> # Replace character
```

```

s      # Select matches for replacement
c      # Change selection
|      # Pipe through sed for replacement

# Global replace example
%      # Select all
s pattern # Select all pattern matches
c replacement<Esc> # Replace all

```

File and Buffer Management

Buffer Operations

```

# Buffer navigation
g a    # Go to alternate buffer
g f    # Go to file under cursor
g F    # Go to file under cursor (new buffer)

# Buffer commands (in command mode)
:buffer <name>    # Switch to buffer
:buffer-next      # Next buffer
:buffer-previous  # Previous buffer
:delete-buffer    # Close current buffer
:write            # Save buffer
:write-all       # Save all buffers
:quit             # Quit
:quit!           # Quit without saving
:write-quit       # Save and quit

```

File Operations

```

# File commands
:edit <file>    # Open file
:new           # New buffer
:write <file>   # Save as file
:cd <dir>       # Change directory
:pwd           # Show current directory

```

Window Management

```
# Window splitting (requires tmux or similar)
# Kakoune doesn't have built-in splits
# Use tmux or terminal multiplexer

# Multiple clients
kak -s main file.txt      # Start session
kak -c main               # Connect new client
```

Configuration

Configuration File

Kakoune configuration is stored in `~/.config/kak/kakrc`:

```
# ~/.config/kak/kakrc

# Basic settings
set-option global tabstop 4
set-option global indentwidth 4
set-option global scrolloff 5
set-option global ui_options ncurses_status_on_top=true
set-option global ui_options ncurses_assistant=cat

# Enable line numbers
add-highlighter global/ number-lines -relative -hlcursor

# Enable matching bracket highlighting
add-highlighter global/ show-matching

# Enable whitespace highlighting
add-highlighter global/ show-whitespaces -tab "→" -tabpad "." -lf "↵" -spc "."

# Color scheme
colorscheme default

# Key mappings
map global normal <c-p> ': fzf-mode<ret>'
map global normal '#' ': comment-line<ret>'
map global normal '<a-#>' ': comment-block<ret>'

# User mode for custom commands
declare-user-mode user
map global normal <space> ': enter-user-mode user<ret>'
```

```

map global user f ': fzf-mode<ret>'
map global user g ': grep '
map global user r ': replace-mode<ret>'

# Auto-pairs
hook global InsertChar \( %{ exec '<a-;>><left>' }
hook global InsertChar \[ %{ exec '<a-;>]<left>' }
hook global InsertChar \{ %{ exec '<a-;>><left>' }
hook global InsertChar \" %{ exec '<a-;>\"<left>' }
hook global InsertChar \' %{ exec '<a-;>\'<left>' }

# Auto-indent
hook global InsertChar \n %{ exec -draft \; K <a-&> }
hook global InsertChar \} %{ exec -draft <a-;> <a-S> 1<a-&> }

# Highlight TODO/FIXME
add-highlighter global/ regex \b(TODO|FIXME|XXX|NOTE)\b 0:default+rb

# File type specific settings
hook global BufSetOption filetype=python %{
    set-option buffer indentwidth 4
    set-option buffer tabstop 4
}

hook global BufSetOption filetype=javascript %{
    set-option buffer indentwidth 2
    set-option buffer tabstop 2
}

hook global BufSetOption filetype=html %{
    set-option buffer indentwidth 2
    set-option buffer tabstop 2
}

# Clipboard integration
map global user y '<a-|>xclip -i -selection clipboard<ret>'
map global user p '!xclip -o -selection clipboard<ret>'

# Save and quit shortcuts
map global user w ': write<ret>'
map global user q ': quit<ret>'
map global user x ': write-quit<ret>'

```

Plugin Management

Kakoune doesn't have a built-in plugin manager, but you can use external tools:

```
# Install plug.kak (plugin manager)
mkdir -p ~/.config/kak/plugins
git clone https://github.com/andreystor/plug.kak.git ~/.config/kak/plugins/plug.kak

# Add to kakrc
source "%val{config}/plugins/plug.kak/rc/plug.kak"
plug "andreystor/plug.kak" noload

# Install plugins
plug "andreystor/fzf.kak"
plug "andreystor/smarttab.kak"
plug "ul/kak-lsp"
plug "alexherbo2/auto-pairs.kak"
plug "alexherbo2/surround.kak"
plug "delapouite/kakoune-buffers"
plug "occivink/kakoune-snippets"
```

Popular Plugins

```
# Essential plugins configuration

# FZF integration
plug "andreystor/fzf.kak" config %{
    map global normal <c-p> ': fzf-mode<ret>'
    map global normal <c-o> ': fzf-file<ret>'
}

# LSP support
plug "ul/kak-lsp" do %{
    cargo install --locked --force --path .
} config %{
    set global lsp_diagnostic_line_error_sign ' '
    set global lsp_diagnostic_line_warning_sign ' '

    define-command ne -docstring 'go to next error/warning from lsp' %{
        lsp-find-error --include-warnings
    }
    define-command pe -docstring 'go to previous error/warning from lsp' %{
        lsp-find-error --previous --include-warnings
    }
    define-command ee -docstring 'go to current error/warning from lsp' %{
        lsp-find-error --include-warnings --previous
        lsp-find-error --include-warnings
    }
}
```

```

map global user l %{: enter-user-mode lsp<ret>} -docstring "LSP mode"
map global insert <tab> '<a-;>: try lsp-snippets-select-next-placeholders catch %{' execu
map global object a '<a-semicolon>lsp-object<ret>' -docstring 'LSP any symbol'
map global object <a-a> '<a-semicolon>lsp-object<ret>' -docstring 'LSP any symbol'
map global object f '<a-semicolon>lsp-object Function Method<ret>' -docstring 'LSP funct
map global object t '<a-semicolon>lsp-object Class Interface Struct<ret>' -docstring 'LS

hook global WinSetOption filetype=(c|cpp|cc|rust|javascript|typescript|python) %{'
    lsp-enable-window
}

}

# Surround text objects
plug "alexherbo2/surround.kak" config %{'
    map global user s ': surround<ret>' -docstring 'surround'
    map global user S ': surround _ _ * *<ret>' -docstring 'surround with *'
}

# Auto-pairs
plug "alexherbo2/auto-pairs.kak" config %{'
    enable-auto-pairs
}

# Snippets
plug "occivink/kakoune-snippets" config %{'
    set-option global snippets_directories "%opt{plug_install_dir}/kakoune-snippets/snippets
}

# Buffer management
plug "delapouite/kakoune-buffers" config %{'
    map global user b ': enter-user-mode buffers<ret>' -docstring 'buffers'
    map global user B ': enter-user-mode -lock buffers<ret>' -docstring 'buffers (lock)'
}

```

Real-World Usage Scenarios

Scenario 1: Refactoring Variable Names

```

# Open file
kak script.py

# Select variable name
w      # Select word

# Select all occurrences

```

```
%      # Select all matches

# Change all occurrences
c      # Change selection
new_variable_name<Esc>

# Save file
:w
```

Scenario 2: Multiple Line Editing

```
# Select multiple lines
x      # Select line
j      # Move down
x      # Extend to next line
j      # Move down
x      # Extend to next line

# Or select lines with pattern
%      # Select all
s^def # Select lines starting with "def"

# Add prefix to all selected lines
I      # Insert at beginning
# Type prefix
<Esc>
```

Scenario 3: Complex Text Manipulation

```
# Select paragraph
<a-a>p # Select around paragraph

# Split into sentences
s\. # Split on periods

# Process each sentence
|capitalize_first_word.sh # Pipe through script

# Or use built-in commands
~      # Switch case of first character
```

Scenario 4: Log File Analysis

```
# Open log file
kak /var/log/application.log

# Select all error lines
%      # Select all
s^.*ERROR.*$ # Select lines containing ERROR

# Copy to new buffer
y      # Yank
:new   # New buffer
p      # Paste

# Analyze timestamps
s\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2} # Select timestamps
```

Advanced Features

Hooks and Automation

```
# Auto-save on focus lost
hook global FocusOut .* %{ write }

# Auto-format on save
hook global BufWritePre .*\.py %{ format }

# Highlight current word
hook global NormalIdle .* %{
    eval -draft %{ try %{
        exec <space><a-i>w <a-k>\A\w+\z<ret>
        add-highlighter window/word regex "\b\Q%val{selection}\E\b" 0:+u
    } catch %{
        rmhl window/word
    } }
}

# Auto-reload files changed externally
hook global FocusIn .* %{ edit! }
```


Custom Commands

```
# Define custom commands
define-command -docstring "duplicate line" duplicate-line %{
    exec 'x<a-s>y<a-o>p'
}

define-command -docstring "sort selections" sort-selections %{
    exec '|sort<ret>'
}

define-command -docstring "remove duplicates" remove-duplicates %{
    exec '|sort -u<ret>'
}

define-command -docstring "count words" count-words %{
    exec '<a-s>|wc -w<ret>'
}

# Map to keys
map global user d ': duplicate-line<ret>'
map global user s ': sort-selections<ret>'
map global user u ': remove-duplicates<ret>'
map global user c ': count-words<ret>'
```

Text Objects

```
# Built-in text objects
<a-i>w # Inner word
<a-a>w # Around word
<a-i>s # Inner sentence
<a-a>s # Around sentence
<a-i>p # Inner paragraph
<a-a>p # Around paragraph
<a-i>( # Inner parentheses
<a-a>( # Around parentheses
<a-i>{ # Inner braces
<a-a>{ # Around braces
<a-i>[ # Inner brackets
<a-a>[ # Around brackets
<a-i>" # Inner quotes
<a-a>" # Around quotes

# Custom text objects
define-command -docstring "select function" select-function %{
```

```

    exec '<a-s>s~def\s+\w+.*?(?~=^def|\z)<ret>'
}

map global object f ': select-function<ret>'

```

Macros and Repetition

```

# Record macro
q<register> # Start recording to register
# ... perform actions ...
q          # Stop recording

# Play macro
@<register> # Execute macro from register
@@         # Repeat last macro

# Repeat last command
.          # Repeat last normal mode command
<a-.>      # Repeat last insert mode command

```

Language-Specific Configurations

Python Development

```

# Python-specific kakrc additions
hook global BufSetOption filetype=python %{
    set-option buffer indentwidth 4
    set-option buffer tabstop 4

    # Python-specific mappings
    map buffer user r ': python-run<ret>'
    map buffer user t ': python-test<ret>'
    map buffer user f ': python-format<ret>'

    # Auto-format with black
    hook buffer BufWritePre .* %{ exec '|black -<ret>' }

    # Highlight Python keywords
    add-highlighter buffer/python-keywords regex \b(def|class|import|from|if|elif|else|for|w
}

define-command python-run %{
    exec '|python3<ret>'
}

```

```

}

define-command python-test %{
    exec '!python3 -m pytest<ret>'
}

define-command python-format %{
    exec '%|black -<ret>'
}

```

Web Development

```

# HTML/CSS/JavaScript configuration
hook global BufSetOption filetype=(html|css|javascript) %{
    set-option buffer indentwidth 2
    set-option buffer tabstop 2

    # Auto-close tags for HTML
    hook buffer InsertChar > %{
        exec -draft '<a-;>h<a-k><[~/][^>]*<ret>a/<c-r>"><left>'
    }
}

# JavaScript-specific
hook global BufSetOption filetype=javascript %{
    map buffer user r ': javascript-run<ret>'
    map buffer user f ': javascript-format<ret>'

    # Format with prettier
    hook buffer BufWritePre .* %{ exec '%|prettier --stdin-filepath %val{buffile}<ret>' }
}

define-command javascript-run %{
    exec '|node<ret>'
}

define-command javascript-format %{
    exec '%|prettier --stdin-filepath %val{buffile}<ret>'
}

```

Go Development

```
# Go-specific configuration
hook global BufSetOption filetype=go %{
    set-option buffer indentwidth 4
    set-option buffer tabstop 4

    map buffer user r ': go-run<ret>'
    map buffer user b ': go-build<ret>'
    map buffer user t ': go-test<ret>'
    map buffer user f ': go-format<ret>'

    # Auto-format with gofmt
    hook buffer BufWritePre .* %{ exec '%|gofmt<ret>' }
}

define-command go-run %{
    exec '!go run %val{buffile}<ret>'
}

define-command go-build %{
    exec '!go build<ret>'
}

define-command go-test %{
    exec '!go test<ret>'
}

define-command go-format %{
    exec '%|gofmt<ret>'
}
```

Integration with External Tools

Git Integration

```
# Git commands
define-command git-status %{
    exec '!git status<ret>'
}

define-command git-add %{
    exec '!git add %val{buffile}<ret>'
}
```

```

define-command git-commit %{
    exec '!git commit<ret>'
}

define-command git-diff %{
    exec '!git diff %val{buffile}<ret>'
}

# Map to user mode
map global user g ': enter-user-mode git<ret>' -docstring 'git'
declare-user-mode git
map global git s ': git-status<ret>' -docstring 'status'
map global git a ': git-add<ret>' -docstring 'add'
map global git c ': git-commit<ret>' -docstring 'commit'
map global git d ': git-diff<ret>' -docstring 'diff'

```

Build System Integration

```

# Build system commands
define-command make %{
    exec '!make<ret>'
}

define-command make-clean %{
    exec '!make clean<ret>'
}

define-command make-install %{
    exec '!make install<ret>'
}

# Map to user mode
map global user m ': enter-user-mode make<ret>' -docstring 'make'
declare-user-mode make
map global make m ': make<ret>' -docstring 'make'
map global make c ': make-clean<ret>' -docstring 'clean'
map global make i ': make-install<ret>' -docstring 'install'

```

Terminal Integration

```
# Terminal commands
define-command terminal %{
    exec '!'$SHELL<ret>'
}

define-command terminal-here %{
    exec '!'cd %sh{dirname "$kak_buffile"} && $SHELL<ret>'
}

map global user t ': terminal<ret>' -docstring 'terminal'
map global user T ': terminal-here<ret>' -docstring 'terminal here'
```

Performance and Optimization

Large File Handling

```
# Optimize for large files
hook global BufCreate .*large.* %{
    set-option buffer readonly true
    rmhl buffer/number-lines
    rmhl buffer/show-matching
}

# Disable expensive features for large files
hook global BufOpenFile .* %{
    eval %sh{
        if [ $(wc -l < "$kak_hook_param") -gt 10000 ]; then
            echo "
                rmhl buffer/number-lines
                rmhl buffer/show-matching
                rmhl buffer/show-whitespaces
            "
        fi
    }
}
```

Memory Optimization

```
# Limit history size
set-option global history_size 100

# Disable undo for large operations
define-command no-undo %{
    set-option buffer history_size 0
}

map global user n ': no-undo<ret>' -docstring 'disable undo'
```

Troubleshooting

Common Issues

```
# Debug mode
kak -d

# Check configuration
:debug info

# Reset to defaults
kak -n # No user config

# Check key mappings
:map

# Check options
:set-option
```

Session Management

```
# List sessions
kak -l

# Kill session
kak -c session_name -e 'kill'

# Kill all sessions
for session in $(kak -l); do
    kak -c "$session" -e 'kill'
done
```

Configuration Issues

```
# Test configuration
kak -f /dev/null # Empty config

# Check for errors
kak -e 'echo %val{client_env_TERM}'

# Backup and reset config
mv ~/.config/kak ~/.config/kak.backup
```

Kakoune offers a unique and powerful approach to text editing through its selection-first philosophy and multiple selections. While it has a learning curve, its orthogonal design and composable commands make it highly effective for complex text manipulation tasks.

Modern Tools

Modern Tools

Modern Linux Development Tools

This chapter is the **companion CLI layer** around editors and workspaces: better search, navigation, diffs, file managers, and prompts. Pair it with a daily editor (Neovim, Helix, Micro, ...) and a session tool ([tmux](#), [Herdr](#), or Zellij below).

For end-to-end recipes, see [Workflows](#).

Why Modern Tools on Linux?

- **Native Performance:** Tools written in Rust and Go provide excellent performance on Linux
- **System Integration:** Deep integration with Linux package managers and system services
- **Development Focus:** Designed with Linux development workflows in mind
- **Resource Efficiency:** Optimized for Linux server environments and containers
- **Open Source:** Most tools are open source and align with Linux philosophy

Suggested “editor kit” install

```
# Debian/Ubuntu names vary slightly (fd-find, batcat)
sudo apt install -y ripgrep fd-find fzf bat git curl
# Optional polish:
# cargo install eza zoxide git-delta starship just
# cargo install --locked lazygit # or use distro package
```

Wire defaults:

```
export EDITOR=nvim # or hx, micro, "zed --wait"
export FZF_DEFAULT_COMMAND='fd --type f --hidden --follow --exclude .git'
export MANPAGER="sh -c 'col -bx | bat -l man -p'"
```

Terminal Multiplexers (map)

Tool	Role in this book
tmux	Full chapter: tmux — portable SSH sessions
Herdr	Full chapter: Herdr — AI agent workspaces
Zellij	Covered below — friendly layouts and defaults

Tool	Role in this book
GNU screen	Brief notes below — legacy availability

GNU screen (legacy multiplexer)

Still appears on minimal systems when `tmux` is missing.

```
sudo apt install -y screen    # or pacman/dnf equivalent
screen -S work                # named session
# Ctrl-a d                    # detach (prefix is Ctrl-a)
screen -r work                # reattach
screen -ls
```

Prefer learning **tmux** for new muscle memory; use `screen` when it is all you have.

Terminal-Based Development Tools

Zellij: Modern Terminal Multiplexer

Zellij is a modern alternative to `tmux` with better defaults and user experience. For agent-heavy workflows prefer [Herdr](#); for maximum portability prefer [tmux](#).

Installation

```
# Cargo
cargo install zellij

# Arch Linux
sudo pacman -S zellij

# Ubuntu/Debian (from GitHub releases)
wget https://github.com/zellij-org/zellij/releases/latest/download/zellij-x86_64-unknown-linux-musl.tar.gz
tar -xzf zellij-x86_64-unknown-linux-musl.tar.gz
sudo mv zellij /usr/local/bin/

# SUSE/openSUSE
sudo zypper install zellij

# Gentoo
sudo emerge app-misc/zellij
```

Basic Usage

```
# Start zellij
zellij

# Key bindings (default)
Ctrl+p n    # New pane
Ctrl+p x    # Close pane
Ctrl+p h/j/k/l # Navigate panes
Ctrl+p r    # Resize mode
Ctrl+p t    # New tab
Ctrl+p q    # Quit

# Sessions
zellij -s session_name # Start named session
zellij ls              # List sessions
zellij a session_name  # Attach to session
```

Configuration

```
// ~/.config/zellij/config.kdl
keybinds {
  normal {
    bind "Alt h" { MoveFocus "Left"; }
    bind "Alt l" { MoveFocus "Right"; }
    bind "Alt j" { MoveFocus "Down"; }
    bind "Alt k" { MoveFocus "Up"; }
    bind "Alt n" { NewPane; }
    bind "Alt x" { CloseFocus; }
    bind "Alt t" { NewTab; }
    bind "Alt 1" { GoToTab 1; }
    bind "Alt 2" { GoToTab 2; }
    bind "Alt 3" { GoToTab 3; }
  }
}

theme "catppuccin-mocha"
default_shell "zsh"
pane_frames false
```

Starship: Cross-Shell Prompt

Starship provides a fast, customizable prompt for any shell.

Installation

```
# Install script
curl -sS https://starship.rs/install.sh | sh

# Cargo
cargo install starship --locked

# Package managers
# Arch Linux
sudo pacman -S starship

# Ubuntu/Debian
sudo apt install starship

# SUSE/openSUSE
sudo zypper install starship

# Gentoo
sudo emerge app-shells/starship
```

Configuration

```
# Add to shell config (~/.bashrc, ~/.zshrc, etc.)
eval "$(starship init bash)" # For bash
eval "$(starship init zsh)"  # For zsh

# ~/.config/starship.toml
format = """
[ (#9A348E)\
$os\
$username\
[ (bg:#DA627D fg:#9A348E)\
$directory\
[ (fg:#DA627D bg:#FCA17D)\
$git_branch\
$git_status\
[ (fg:#FCA17D bg:#86BBD8)\
$c\
$elixir\
$elm\
$golang\
$gradle\
$haskell\
```

```

$java\
$julia\
$nodejs\
$nim\
$rust\
$scala\
[] (fg:#86BBD8 bg:#06969A)\
$docker_context\
[] (fg:#06969A bg:#33658A)\
$time\
[ ] (fg:#33658A)\
""

[os]
disabled = false
style = "bg:#9A348E"

[username]
show_always = true
style_user = "bg:#9A348E"
style_root = "bg:#9A348E"
format = '[ $user ] ($style)'
disabled = false

[directory]
style = "bg:#DA627D"
format = "[ $path ] ($style)"
truncation_length = 3
truncation_symbol = ".../"

[git_branch]
symbol = ""
style = "bg:#FCA17D"
format = '[ $symbol $branch ] ($style)'

[git_status]
style = "bg:#FCA17D"
format = '[ $all_status $ahead_behind ] ($style)'

[nodejs]
symbol = ""
style = "bg:#86BBD8"
format = '[ $symbol ($version) ] ($style)'

[rust]
symbol = ""

```

```

style = "bg:#86BBD8"
format = '[ $symbol ($version) ]($style)'

[golang]
symbol = ""
style = "bg:#86BBD8"
format = '[ $symbol ($version) ]($style)'

[python]
symbol = ""
style = "bg:#86BBD8"
format = '[ $symbol ($version) ]($style)'

[time]
disabled = false
time_format = "%R"
style = "bg:#33658A"
format = '[ $time ]($style)'

```

Zoxide: Smart Directory Navigation

Zoxide is a smarter cd command that learns your habits.

Installation

```

# Install script
curl -sS https://raw.githubusercontent.com/ajeetdsouza/zoxide/main/install.sh | bash

# Cargo
cargo install zoxide --locked

# Package managers
sudo pacman -S zoxide # Arch
sudo zypper install zoxide # SUSE/openSUSE
sudo emerge app-shells/zoxide # Gentoo

```

Setup

```

# Add to shell config
eval "$(zoxide init bash)" # For bash
eval "$(zoxide init zsh)" # For zsh

```


Usage

```
# Navigate to directories
z foo      # cd to highest ranked directory matching foo
z foo bar  # cd to highest ranked directory matching foo and bar
zi foo     # cd with interactive selection

# Add directories
zoxide add /path/to/directory

# Query database
zoxide query foo
zoxide query --list
```

Eza: Modern ls Replacement

Eza is a modern replacement for ls with better defaults and features.

Installation

```
# Cargo
cargo install eza

# Arch Linux
sudo pacman -S eza

# Ubuntu/Debian
sudo apt install eza

# SUSE/openSUSE
sudo zypper install eza

# Gentoo
sudo emerge sys-apps/eza
```

Usage

```
# Basic usage
eza          # List files
eza -l       # Long format
eza -la      # Long format with hidden files
eza -T       # Tree view
eza -T -L 2  # Tree view with depth limit
```

```
eza --git          # Show git status
eza --icons        # Show file type icons

# Useful aliases
alias ls='eza'
alias ll='eza -l'
alias la='eza -la'
alias lt='eza -T'
alias lg='eza -l --git'
```

Bat: Better Cat

Bat is a cat clone with syntax highlighting and Git integration.

Installation

```
# Package managers
sudo apt install bat      # Ubuntu/Debian
sudo pacman -S bat        # Arch Linux
sudo zypper install bat   # SUSE/openSUSE
sudo emerge sys-apps/bat  # Gentoo
cargo install bat         # Cargo
```

Usage

```
# Basic usage
bat file.txt          # View file with syntax highlighting
bat -n file.txt       # Show line numbers
bat -A file.txt       # Show all characters
bat --theme=GitHub file.txt # Use specific theme

# Useful aliases
alias cat='bat'
alias less='bat'

# Integration with other tools
export MANPAGER="sh -c 'col -bx | bat -l man -p'"
```

Ripgrep: Fast Text Search

Ripgrep (rg) is a fast grep alternative with better defaults.

Installation

```
# Package managers
sudo apt install ripgrep      # Ubuntu/Debian
sudo pacman -S ripgrep       # Arch Linux
sudo zypper install ripgrep  # SUSE/openSUSE
sudo emerge sys-apps/ripgrep # Gentoo
cargo install ripgrep        # Cargo
```

Usage

```
# Basic search
rg "pattern"           # Search in current directory
rg "pattern" /path     # Search in specific path
rg -i "pattern"        # Case insensitive
rg -w "word"           # Whole word search
rg -v "pattern"        # Invert match

# File type filtering
rg -t py "pattern"     # Search only Python files
rg -T py "pattern"     # Exclude Python files
rg --type-list         # List available file types

# Advanced options
rg -n "pattern"        # Show line numbers
rg -C 3 "pattern"      # Show 3 lines of context
rg -A 3 -B 3 "pattern" # Show 3 lines after and before
rg --hidden "pattern"  # Search hidden files
rg --no-ignore "pattern" # Don't respect .gitignore

# Regular expressions
rg "\d{3}-\d{2}-\d{4}" # Phone number pattern
rg "^function.*\{"     # Function definitions
```

Fzf: Fuzzy Finder

fzf is the interactive filter that turns huge lists into something you can pick with two keystrokes. It is the glue between **fd/rg** and your editor.

Installation

```
sudo apt install fzf          # Debian/Ubuntu
sudo pacman -S fzf           # Arch
sudo dnf install fzf         # Fedora
sudo zypper install fzf      # openSUSE
cargo install fzf            # if needed
```

Shell keybindings (bash):

```
# Debian package often provides:
# /usr/share/doc/fzf/examples/key-bindings.bash
source /usr/share/doc/fzf/examples/key-bindings.bash 2>/dev/null || true
# Ctrl-R → fuzzy history, Ctrl-T → fuzzy files (when configured)
```

Editor integration patterns

```
# Open file in $EDITOR
export FZF_DEFAULT_COMMAND='fd --type f --hidden --follow --exclude .git'
export FZF_DEFAULT_OPTS='--height 40% --layout=reverse --border --preview "bat --color=always'

fe() {
    local file
    file=$(fzf) || return
    ${EDITOR:-nvim} "$file"
}

# Search file contents then open at line (requires rg)
feg() {
    local out file line
    out=$(rg -n --hidden --glob '!.git' "${1:-.}" 2>/dev/null | fzf) || return
    file=${out%*: *}
    line=$(printf '%s\n' "$out" | cut -d: -f2)
    ${EDITOR:-nvim} +"$line" "$file"
}
```

Inside Neovim/LazyVim, Telescope or fzf-lua often wraps the same ideas. Inside Helix, use Space-f pickers; still keep shell `fe` for any directory.

Lazygit: Terminal Git UI

lazygit is a keyboard-driven Git TUI. It pairs well with any editor: stage hunks here, press `e` (or your configured key) to open the file in `$EDITOR`.

```
# Install (one of)
sudo pacman -S lazygit
go install github.com/jesseduffield/lazygit@latest
# binary releases: https://github.com/jesseduffield/lazygit/releases
```

```
cd ~/src/app
lazygit
# Common: space=stage, c=commit, p=pull, P=push, / = search
```

LazyVim users often have <leader>gg bound to lazygit. In tmux/Herdr, give lazygit its own pane or window.

Yazi: Terminal File Manager (modern)

Yazi is a fast async terminal file manager (Rust) with previews—useful beside Helix/Neovim when you want browsing without leaving the TUI.

```
cargo install --locked yazi-fm yazi-cli
# or check distro packages / https://yazi-rs.github.io/

ya # or yazi
# Open selection in $EDITOR; use as a project springboard
```

Alternatives already covered below: **nnn**, **broot**.

Fd: Better Find

Fd is a simple, fast alternative to find with better defaults.

Installation

```
# Package managers
sudo apt install fd-find # Ubuntu/Debian (command is fdfind)
sudo pacman -S fd        # Arch Linux
sudo zypper install fd    # SUSE/openSUSE
sudo emerge sys-apps/fd   # Gentoo
cargo install fd-find     # Cargo
```

Usage

```
# Basic usage
fd pattern                # Find files/directories matching pattern
fd -t f pattern           # Find only files
fd -t d pattern           # Find only directories
fd -e py                  # Find Python files
fd -E node_modules pattern # Exclude node_modules

# Advanced options
fd -H pattern             # Include hidden files
fd -I pattern             # Don't respect .gitignore
fd -x command             # Execute command on results
fd pattern -x ls -la      # Execute ls -la on each result

# Useful aliases
alias find='fd'
```

Delta: Better Git Diff

Delta provides syntax highlighting and better formatting for git diffs.

Installation

```
# Package managers
sudo apt install git-delta # Ubuntu/Debian
sudo pacman -S git-delta  # Arch Linux
sudo zypper install git-delta # SUSE/openSUSE
sudo emerge dev-util/git-delta # Gentoo
cargo install git-delta     # Cargo
```

Configuration

```
# Add to ~/.gitconfig
[core]
  pager = delta

[interactive]
  diffFilter = delta --color-only

[delta]
  navigate = true
  light = false
```

```

    side-by-side = true
    line-numbers = true
    syntax-theme = Dracula

[merge]
    conflictstyle = diff3

[diff]
    colorMoved = default

```

File Management Tools

Broot: Interactive Directory Navigation

Broot provides an interactive way to navigate and manipulate directory trees.

Installation

```

# Cargo
cargo install broot

# Package managers
sudo pacman -S broot      # Arch Linux
sudo zypper install broot  # SUSE/openSUSE
sudo emerge app-misc/broot # Gentoo

# Install shell integration
broot --install

```

Usage

```

# Start broot
br                                     # Navigate current directory
br /path/to/directory                # Navigate specific directory

# Key bindings
Ctrl+h                               # Toggle hidden files
Ctrl+f                               # Search files
Ctrl+d                               # Delete file/directory
Ctrl+c                               # Copy path
Enter                                # Open file/enter directory
Alt+Enter                            # Open in external editor

```

Nnn: Terminal File Manager

Nnn is a fast, feature-rich terminal file manager.

Installation

```
# Package managers
sudo apt install nnn      # Ubuntu/Debian
sudo pacman -S nnn        # Arch Linux
sudo zypper install nnn   # SUSE/openSUSE
sudo emerge app-misc/nnn  # Gentoo

# From source
git clone https://github.com/jarun/nnn.git
cd nnn
make
sudo make install
```

Configuration

```
# Environment variables
export NNN_PLUG='f:finder;o:fzopen;p:mocplay;d:diffs;t:nmount;v:imgview'
export NNN_FIFO='/tmp/nnn.fifo'
export NNN_COLORS='2136'
export NNN_FCOLORS='c1e2272e006033f7c6d6abc4'

# Aliases
alias n='nnn -de'
alias N='sudo nnn -dH'
```

Usage

```
# Basic navigation
j/k or ↑/↓      # Navigate files
h/l or ←/→      # Navigate directories
Enter           # Open file/enter directory
q               # Quit
?               # Help

# File operations
d               # Delete
r               # Rename
c               # Copy
v               # Move
```



```
s          # Select files
a          # Select all
A          # Invert selection

# Advanced features
;          # Plugin menu
!          # Shell
e          # Edit in $EDITOR
p          # Show file info
```

Development Environment Tools

Direnv: Environment Management

Direnv automatically loads and unloads environment variables based on directory.

Installation

```
# Package managers
sudo apt install direnv    # Ubuntu/Debian
sudo pacman -S direnv      # Arch Linux
sudo zypper install direnv # SUSE/openSUSE
sudo emerge dev-util/direnv # Gentoo

# Add to shell config
eval "$(direnv hook bash)" # For bash
eval "$(direnv hook zsh)"  # For zsh
```

Usage

```
# Create .envrc file in project directory
echo 'export DATABASE_URL="postgres://localhost/mydb"' > .envrc
echo 'export DEBUG=true' >> .envrc

# Allow the .envrc file
direnv allow

# The environment variables are now loaded when you cd into the directory
# and unloaded when you leave
```

Advanced .envrc Examples

```
# Python virtual environment
layout python3
pip install -r requirements.txt

# Node.js version management
use node 18

# Go module
export GOPATH=$PWD/.go
export PATH=$GOPATH/bin:$PATH

# Docker development
export COMPOSE_FILE=docker-compose.dev.yml
export DOCKER_BUILDKIT=1

# Load from .env file
dotenv

# Path manipulation
PATH_add bin
PATH_add scripts
```

Mise: Runtime Version Manager

Mise (formerly rtx) manages multiple runtime versions for different languages.

Installation

```
# Install script
curl https://mise.run | sh

# Cargo
cargo install mise

# Package managers
sudo pacman -S mise      # Arch Linux
sudo zypper install mise  # SUSE/openSUSE
# For Gentoo, install via cargo or from source
```

Setup

```
# Add to shell config
echo 'eval "$(mise activate bash)"' >> ~/.bashrc # For bash
echo 'eval "$(mise activate zsh)"' >> ~/.zshrc   # For zsh
```

Usage

```
# Install runtimes
mise install node@18      # Install Node.js 18
mise install python@3.11  # Install Python 3.11
mise install go@1.26      # Install a current stable Go (or: mise install go@latest)

# Set global versions
mise global node@18
mise global python@3.11

# Set local versions (creates .mise.toml)
mise local node@16
mise local python@3.9

# List available versions
mise list-remote node
mise list-remote python

# List installed versions
mise list
```

Configuration

```
# .mise.toml
[tools]
node = "18"
python = "3.11"
go = "1.21"

[env]
NODE_ENV = "development"
DEBUG = "true"

[tasks.dev]
run = "npm run dev"
description = "Start development server"
```

```
[tasks.test]
run = "npm test"
description = "Run tests"
```

Just: Command Runner

Just is a command runner similar to make but with better syntax.

Installation

```
# Cargo
cargo install just

# Package managers
sudo pacman -S just      # Arch Linux
sudo zypper install just  # SUSE/openSUSE
sudo emerge sys-devel/just # Gentoo

# Install script
curl --proto 'https' --tlsv1.2 -sSf https://just.systems/install.sh | bash -s -- --to ~/bin
```

Usage

Create a justfile:

```
# Default recipe
default:
    @just --list

# Development server
dev:
    npm run dev

# Run tests
test:
    npm test

# Build project
build:
    npm run build

# Deploy to production
deploy: build
    rsync -av dist/ user@server:/var/www/
```

```

# Clean build artifacts
clean:
    rm -rf dist/
    rm -rf node_modules/

# Install dependencies
install:
    npm install

# Format code
fmt:
    prettier --write .
    eslint --fix .

# Docker commands
docker-build:
    docker build -t myapp .

docker-run:
    docker run -p 3000:3000 myapp

# Database commands
db-migrate:
    npx prisma migrate dev

db-seed:
    npx prisma db seed

# Variables
port := "3000"
env := "development"

# Recipe with parameters
serve port=port:
    NODE_ENV={{env}} npm start -- --port {{port}}

# Conditional recipes
backup:
    #!/usr/bin/env bash
    if [ -f "database.db" ]; then
        cp database.db "database.db.backup.$(date +%Y%m%d_%H%M%S)"
        echo "Database backed up"
    else
        echo "No database file found"
    fi

```

```
# Run recipes
just                                # Run default recipe
just dev                           # Run dev recipe
just serve 8080                     # Run serve with custom port
just --list                         # List all recipes
```

Text Processing Tools

Jq: JSON Processor

Jq is a lightweight command-line JSON processor.

Installation

```
# Package managers
sudo apt install jq           # Ubuntu/Debian
sudo pacman -S jq            # Arch Linux
sudo zypper install jq       # SUSE/openSUSE
sudo emerge app-misc/jq      # Gentoo
```

Usage

```
# Basic usage
echo '{"name": "John", "age": 30}' | jq '.'
echo '{"name": "John", "age": 30}' | jq '.name'
echo ' [{"name": "John"}, {"name": "Jane"} ]' | jq '[0].name'

# Filter arrays
echo '[1,2,3,4,5]' | jq '.[0] | select(. > 3)'

# Transform data
echo '{"users": [{"name": "John", "age": 30}, {"name": "Jane", "age": 25}]}' | \
jq '.users | map({name, adult: (.age >= 18)})'

# Real-world examples
curl -s https://api.github.com/users/octocat | jq '.name, .public_repos'
docker inspect container_name | jq '[0].NetworkSettings.IPAddress'
kubectl get pods -o json | jq '.items[].metadata.name'
```

Yq: YAML Processor

Yq is like jq but for YAML files.

Installation

```
# Go version (recommended)
go install github.com/mikefarah/yq/v4@latest

# Package managers
sudo apt install yq          # Ubuntu/Debian
sudo zypper install yq       # SUSE/openSUSE
sudo emerge app-misc/yq      # Gentoo
```

Usage

```
# Basic usage
yq '.name' config.yaml
yq '.database.host' config.yaml
yq '.services[0].name' docker-compose.yml

# Modify YAML
yq '.database.port = 5432' config.yaml
yq '.services[0].image = "nginx:latest"' docker-compose.yml

# Multiple files
yq eval-all '. as $item ireduce ({}; . * $item)' file1.yaml file2.yaml
```

Sd: Sed Alternative

Sd is a more intuitive alternative to sed for find and replace operations.

Installation

```
# Cargo
cargo install sd

# Package managers
sudo pacman -S sd          # Arch Linux
sudo zypper install sd      # SUSE/openSUSE
sudo emerge sys-apps/sd     # Gentoo
```

Usage

```
# Basic find and replace
sd 'old_text' 'new_text' file.txt
sd '\d{4}-\d{2}-\d{2}' 'DATE' file.txt

# In-place editing
sd -i 'old_text' 'new_text' file.txt

# Preview changes
sd 'old_text' 'new_text' file.txt --preview

# Multiple files
sd 'old_text' 'new_text' *.txt
```

Monitoring and System Tools

Htop: Process Viewer

Htop is an interactive process viewer and system monitor.

Installation

```
# Package managers
sudo apt install htop      # Ubuntu/Debian
sudo pacman -S htop        # Arch Linux
sudo zypper install htop   # SUSE/openSUSE
sudo emerge sys-process/htop # Gentoo
```

Usage

```
# Start htop
htop

# Key bindings
F1          # Help
F2          # Setup
F3          # Search
F4          # Filter
F5          # Tree view
F6          # Sort by
F9          # Kill process
```



```
F10                                # Quit
```

Bottom: System Monitor

Bottom (btm) is a cross-platform system monitor written in Rust.

Installation

```
# Cargo
cargo install bottom

# Package managers
sudo pacman -S bottom      # Arch Linux
sudo zypper install bottom # SUSE/openSUSE
sudo emerge sys-process/bottom # Gentoo

# Debian/Ubuntu
curl -LO https://github.com/ClementTsang/bottom/releases/download/0.9.6/bottom_0.9.6_amd64.d
sudo dpkg -i bottom_0.9.6_amd64.deb
```

Usage

```
# Start bottom
btm

# Key bindings
?           # Help
q           # Quit
/           # Search
Tab         # Next widget
Shift+Tab   # Previous widget
+/-         # Zoom in/out
```

Procs: Process Information

Procs is a modern replacement for ps with better output formatting.

Installation

```
# Cargo
cargo install procs

# Package managers
sudo pacman -S procs      # Arch Linux
sudo zypper install procs # SUSE/openSUSE
sudo emerge sys-process/procs # Gentoo
```

Usage

```
# Basic usage
procs                # List all processes
procs firefox        # Find processes matching "firefox"
procs --tree          # Tree view
procs --watch         # Watch mode
procs --sortd cpu     # Sort by CPU usage (descending)
```

Linux-Specific Integration

Systemd Integration

Many modern tools can be integrated with systemd for better Linux system management:

```
# Create user service for auto-updating tools
mkdir -p ~/.config/systemd/user

# Auto-update mise tools
cat > ~/.config/systemd/user/mise-update.service << 'EOF'
[Unit]
Description=Update mise tools
After=network.target

[Service]
Type=oneshot
ExecStart=/usr/local/bin/mise upgrade
User=%i
EOF

cat > ~/.config/systemd/user/mise-update.timer << 'EOF'
[Unit]
Description=Update mise tools daily
```

```

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
EOF

# Enable the timer
systemctl --user enable mise-update.timer
systemctl --user start mise-update.timer

```

Linux Distribution Package Management

```

# Create a script to install modern tools across different distros
#!/bin/bash
# install-modern-tools.sh

detect_distro() {
    if [ -f /etc/os-release ]; then
        . /etc/os-release
        echo $ID
    elif [ -f /etc/redhat-release ]; then
        echo "rhel"
    elif [ -f /etc/debian_version ]; then
        echo "debian"
    else
        echo "unknown"
    fi
}

install_tools() {
    local distro=$(detect_distro)

    case $distro in
        ubuntu|debian)
            sudo apt update
            sudo apt install -y ripgrep fd-find bat eza zoxide starship direnv
            ;;
        arch|manjaro)
            sudo pacman -S ripgrep fd bat eza zoxide starship direnv
            ;;
        fedora)
            sudo dnf install -y ripgrep fd-find bat eza zoxide starship direnv
            ;;
    esac
}

```

```

    opensuse*)suse)
        sudo zypper install -y ripgrep fd bat eza zoxide starship direnv
        ;;
    gentoo)
        sudo emerge sys-apps/ripgrep sys-apps/fd sys-apps/bat sys-apps/eza \
            app-shells/zoxide app-shells/starship dev-util/direnv
        ;;
    *)
        echo "Unsupported distribution: $distro"
        echo "Please install tools manually using cargo or from source"
        ;;
esac
}

install_tools

```

Linux Desktop Integration

```

# Create desktop entries for terminal-based tools
mkdir -p ~/.local/share/applications

# Helix desktop entry
cat > ~/.local/share/applications/helix.desktop << 'EOF'
[Desktop Entry]
Name=Helix
Comment=A post-modern modal text editor
Exec=gnome-terminal -- hx %F
Icon=text-editor
Terminal=false
Type=Application
Categories=Development;TextEditor;
MimeType=text/plain;text/x-chdr;text/x-csrc;text/x-c++hdr;text/x-c++src;text/x-java;text/x-d
EOF

# Micro desktop entry
cat > ~/.local/share/applications/micro.desktop << 'EOF'
[Desktop Entry]
Name=Micro
Comment=A modern and intuitive terminal-based text editor
Exec=gnome-terminal -- micro %F
Icon=text-editor
Terminal=false
Type=Application
Categories=Development;TextEditor;
MimeType=text/plain;text/x-chdr;text/x-csrc;text/x-c++hdr;text/x-c++src;text/x-java;text/x-d

```

```
EOF
```

```
# Update desktop database
update-desktop-database ~/.local/share/applications
```

Integration Examples

Complete Development Setup

```
# ~/.zshrc or ~/.bashrc

# Modern tools
eval "$(starship init zsh)"
eval "$(zoxide init zsh)"
eval "$(direnv hook zsh)"
eval "$(mise activate zsh)"

# Aliases
alias ls='eza'
alias ll='eza -l'
alias la='eza -la'
alias lt='eza -T'
alias cat='bat'
alias find='fd'
alias grep='rg'
alias ps='procs'
alias top='btm'

# Functions
function fzf-edit() {
    local file
    file=$(fd --type f | fzf --preview 'bat --color=always {}')
    if [[ -n $file ]]; then
        $EDITOR "$file"
    fi
}

function fzf-cd() {
    local dir
    dir=$(fd --type d | fzf --preview 'eza -T {} | head -50')
    if [[ -n $dir ]]; then
        cd "$dir"
    fi
}
```

```
# Key bindings
bindkey '^F' fzf-edit
bindkey '^G' fzf-cd
```

Project Template with Modern Tools

```
# project-setup.sh
#!/bin/bash

PROJECT_NAME=$1
if [[ -z $PROJECT_NAME ]]; then
    echo "Usage: $0 <project-name>"
    exit 1
fi

# Create project directory
mkdir "$PROJECT_NAME"
cd "$PROJECT_NAME"

# Initialize git
git init

# Create .envrc for direnv
cat > .envrc << 'EOF'
export PROJECT_NAME=$(basename $PWD)
export NODE_ENV=development
export DEBUG=true

# Add project bin to PATH
PATH_add bin
PATH_add scripts

# Load .env if it exists
dotenv_if_exists
EOF

# Create .mise.toml for runtime management
cat > .mise.toml << 'EOF'
[tools]
node = "18"
python = "3.11"

[env]
NODE_ENV = "development"
```

```

[tasks.dev]
run = "npm run dev"
description = "Start development server"

[tasks.test]
run = "npm test"
description = "Run tests"

[tasks.build]
run = "npm run build"
description = "Build project"
EOF

# Create justfile for task running
cat > justfile << 'EOF'
# List available recipes
default:
    @just --list

# Install dependencies
install:
    npm install

# Start development server
dev:
    npm run dev

# Run tests
test:
    npm test

# Build project
build:
    npm run build

# Format code
fmt:
    prettier --write .
    eslint --fix .

# Clean build artifacts
clean:
    rm -rf dist/
    rm -rf node_modules/
EOF

```

```
# Create .gitignore
cat > .gitignore << 'EOF'
node_modules/
dist/
.env
.DS_Store
*.log
EOF

# Allow direnv
direnv allow

echo "Project $PROJECT_NAME created successfully!"
echo "Run 'cd $PROJECT_NAME && just install' to get started"
```

Linux-Specific Troubleshooting

Permission Issues

```
# Fix common permission issues with modern tools
# Ensure tools are executable
chmod +x ~/.local/bin/*

# Fix cargo-installed tools permissions
chmod +x ~/.cargo/bin/*

# Add to PATH if needed
echo 'export PATH="$HOME/.local/bin:$HOME/.cargo/bin:$PATH"' >> ~/.bashrc
```

Package Manager Conflicts

```
# Handle conflicts between package managers
# Remove system package before installing via cargo
sudo apt remove ripgrep # Remove system version
cargo install ripgrep    # Install latest via cargo

# Or use alternatives command on Debian/Ubuntu
sudo update-alternatives --install /usr/bin/rg rg /usr/bin/rg 10
sudo update-alternatives --install /usr/bin/rg rg ~/.cargo/bin/rg 20
```


Performance Optimization for Linux

```
# Optimize tools for Linux systems
# Set environment variables for better performance
export RIPGREP_CONFIG_PATH="$HOME/.ripgrepreprc"
export BAT_THEME="Dracula"
export FZF_DEFAULT_COMMAND='fd --type f --hidden --follow --exclude .git'

# Create ripgrep config for Linux development
cat > ~/.ripgrepreprc << 'EOF'
# Search hidden files but not .git
--hidden
--glob=! .git/*

# Follow symbolic links
--follow

# Smart case search
--smart-case

# Show line numbers
--line-number

# Limit search depth for performance
--max-depth=10
EOF
```

Container Integration

```
# Use modern tools in Docker containers
# Dockerfile example
FROM ubuntu:22.04

RUN apt-get update && apt-get install -y \
    curl \
    git \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Install Rust for cargo-based tools
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
ENV PATH="/root/.cargo/bin:${PATH}"

# Install modern tools
RUN cargo install ripgrep fd-find bat eza zoxide starship
```

```
# Set up shell with modern tools
RUN echo 'eval "$(starship init bash)"' >> ~/.bashrc
RUN echo 'eval "$(zoxide init bash)"' >> ~/.bashrc
RUN echo 'alias ls=eza' >> ~/.bashrc
RUN echo 'alias cat=bat' >> ~/.bashrc
```

Putting it together with editors

Goal	Tooling
Jump to any file	<code>fd + fzf + \$EDITOR</code>
Jump to content	<code>rg + fzf + \$EDITOR +line</code>
Review changes	<code>delta + lazygit</code>
Long SSH session	tmux
Multiple AI agents	Herdr
Pretty shell context	<code>starship + zoxide + eza</code>
Full recipes	Workflows

These modern tools enhance the Linux development experience with better defaults and performance—especially when paired with a deliberate editor and workspace choice from the rest of this book.

Tmux

tmux

tmux: Terminal Multiplexer

tmux is the default long-session tool for Linux servers and SSH. It keeps shells, editors, and builds running after disconnects, and it splits one terminal into windows and panes you can script and reattach from anywhere.

If you learn only one multiplexer, learn this one.

Why tmux

- **Detach / reattach** — close SSH without killing jobs
- **Windows and panes** — layouts for editor + shell + logs
- **Scriptable** — create sessions from shell scripts and automation
- **Ubiquitous** — packages on virtually every distro
- **Pairs with any editor** — nvim, Helix, nano, Emacs **-nw**

Related tools: **GNU screen** (older), **Zellij** (friendlier defaults), **Herdr** (agent-aware). See the intro on [Terminal Workspaces](#) and the Herdr chapter.

Installation

```
# Debian / Ubuntu
sudo apt update && sudo apt install -y tmux

# Fedora
sudo dnf install -y tmux

# Arch
sudo pacman -S tmux

# openSUSE
sudo zypper install tmux

# Alpine
sudo apk add tmux

# Nix
nix profile install nixpkgs#tmux
# or in a flake/devShell: pkgs.tmux
```

```
# macOS (if you dual-boot workflows)
brew install tmux
```

Verify:

```
tmux -V
```

Mental model

```
tmux server (background)
  session "work"
    window 0: editor
      pane 0: nvim
      pane 1: shell
    window 1: logs
      pane 0: journalctl -f
```

Object	What it is
Server	Background process owning all sessions
Session	Named group you attach to (work , prod)
Window	Full-screen tab inside a session
Pane	Split rectangle inside a window

Quick start lab

```
# Create a named session
tmux new -s work

# Inside tmux - prefix is Ctrl-b by default
# Split vertical / horizontal
Ctrl-b %
Ctrl-b "

# Move between panes
Ctrl-b o
Ctrl-b ←/→/↑/↓

# New window, rename, switch
Ctrl-b c
Ctrl-b ,
Ctrl-b n / Ctrl-b p
Ctrl-b 0..9
```

```
# Detach (jobs keep running)
Ctrl-b d

# List and reattach
tmux ls
tmux attach -t work
```

Essential keybindings

Prefix is shown as C-b (hold Ctrl, press b, release, then press the command key).

Sessions

Keys	Action
C-b d	Detach
C-b s	Session picker
C-b \$	Rename session

Windows

Keys	Action
C-b c	New window
C-b ,	Rename window
C-b n / C-b p	Next / previous
C-b w	Window list
C-b &	Kill window
C-b f	Find window by name

Panes

Keys	Action
C-b %	Split left/right
C-b "	Split top/bottom
C-b o	Next pane
C-b ;	Last pane
C-b x	Kill pane
C-b z	Zoom pane (toggle fullscreen)
C-b { / C-b }	Swap panes
C-b Space	Cycle layouts
C-b C-o	Rotate panes

Keys	Action
C-b q	Show pane numbers (then type number)

Copy mode (scrollback)

Keys	Action
C-b [	Enter copy mode
q	Quit copy mode
C-b]	Paste tmux buffer

With vi mode (recommended):

```
set -g mode-keys vi
```

Then in copy mode: v begin selection, y copy (depending on version/config—see config section).

Command-line interface

You can drive tmux without keybindings—ideal for scripts and muscle memory via shell aliases.

```
# Sessions
tmux new -s api -d                # create detached
tmux new -s api -c ~/src/api      # start in directory
tmux attach -t api
tmux attach -t api -d            # attach and detach others
tmux switch -t api
tmux rename-session -t api backend
tmux kill-session -t api
tmux kill-server                  # nuclear: all sessions

# Windows
tmux new-window -t api -n tests
tmux select-window -t api:tests
tmux kill-window -t api:tests

# Panes
tmux split-window -h -t api
tmux split-window -v -t api
tmux select-pane -t api:0.1
tmux send-keys -t api:0.1 'htop' Enter

# Info
tmux list-sessions
```



```
tmux list-windows -t api
tmux list-panes -t api
tmux display-message -p '#S #I #P'
```

Send keys and automation

```
# Start a dev layout detached
SESSION=dev
tmux has-session -t "$SESSION" 2>/dev/null || {
    tmux new-session -d -s "$SESSION" -n editor -c "$HOME/proj"
    tmux send-keys -t "$SESSION:editor" 'nvim' Enter
    tmux split-window -h -t "$SESSION:editor" -c "$HOME/proj"
    tmux send-keys -t "$SESSION:editor.1" 'git status' Enter
    tmux new-window -t "$SESSION" -n run -c "$HOME/proj"
    tmux send-keys -t "$SESSION:run" 'just dev' Enter
}
tmux attach -t "$SESSION"
```

Configuration (~/.tmux.conf)

Reload after edits:

```
tmux source-file ~/.tmux.conf
# or inside tmux: C-b : source-file ~/.tmux.conf
```

Sensible starter config

```
# ~/.tmux.conf - practical defaults

# Terminal & color
set -g default-terminal "tmux-256color"
set -ag terminal-overrides ",xterm-256color:RGB"
set -g history-limit 50000
set -g mouse on
set -g base-index 1
setw -g pane-base-index 1
set -g renumber-windows on
set -g set-clipboard on

# Timing
set -sg escape-time 10
set -g focus-events on
```

```

# Vi keys
set -g mode-keys vi
set -g status-keys vi

# Prefix: keep Ctrl-b or switch to Ctrl-a (uncomment)
# unbind C-b
# set -g prefix C-a
# bind C-a send-prefix

# Splits in current path
bind | split-window -h -c "#{pane_current_path}"
bind - split-window -v -c "#{pane_current_path}"
unbind '"'
unbind %

# Pane move (vim-like)
bind h select-pane -L
bind j select-pane -D
bind k select-pane -U
bind l select-pane -R

# Resize
bind -r H resize-pane -L 5
bind -r J resize-pane -D 5
bind -r K resize-pane -U 5
bind -r L resize-pane -R 5

# Reload config
bind r source-file ~/.tmux.conf \; display-message "reloaded"

# Copy mode (vi)
bind -T copy-mode-vi v send -X begin-selection
bind -T copy-mode-vi y send -X copy-selection-and-cancel

# Status
set -g status-interval 5
set -g status-left-length 40
set -g status-left "[bold]#S #[fg=colour8]| "
set -g status-right "#{?client_prefix,PREFIX ,}%Y-%m-%d %H:%M"
setw -g window-status-current-format "[bold]#I:#W*"
setw -g window-status-format "#I:#W"

```

Nested tmux (SSH jump hosts)

When you run tmux locally and on a remote:

```
# Local: prefix C-b
# Remote: send prefix with C-b C-b (default) or rebind remote to C-a
```

Or use different prefixes per machine and document them.

Nested editors and key conflicts

Symptom	Likely cause
C-b does nothing useful in Vim	You pressed tmux prefix; press again or wait
Esc feels slow in Vim	Raise <code>escape-time</code> was high; set to 10
Colors wrong in nvim	<code>default-terminal</code> / <code>terminal-overrides</code>
Mouse selects but cannot paste	Copy-mode vs terminal clipboard; enable <code>set-clipboard</code> and OSC52-aware terminal

Neovim tip: avoid binding C-b without checking tmux.

Helix/Zed: fewer conflicts; still test zoom (C-b z) and copy mode.

SSH workflows

```
# Reattach or create
ssh host 'tmux new -A -s main'

# From local alias
alias tmain='ssh -t host tmux new -A -s main'
```

`new -A` attaches if the session exists, otherwise creates it.

Keepalive

```
# ~/.ssh/config
Host *
    ServerAliveInterval 30
    ServerAliveCountMax 4
```

tmux survives SSH death; keepalive reduces nuisance disconnects.

Plugins (optional)

TPM is the common plugin manager:

```
git clone https://github.com/tmux-plugins/tpm ~/.tmux/plugins/tpm

# end of ~/.tmux.conf
set -g @plugin 'tmux-plugins/tpm'
set -g @plugin 'tmux-plugins/tmux-sensible'
set -g @plugin 'tmux-plugins/tmux-resurrect' # save/restore sessions
set -g @plugin 'tmux-plugins/tmux-continuum' # auto-save
set -g @plugin 'tmux-plugins/tmux-yank'      # clipboard helpers

run '~/.tmux/plugins/tpm/tpm'
```

Install plugins: C-b I (capital i).

Resurrect restores windows after reboot *if you save*; it is not magic. Critical production jobs still need systemd or proper process supervision.

Pairing with editors

Neovim / Vim

```
tmux new -s code -c ~/src/app
# pane 0
nvim
# C-b %
# pane 1
cargo test -- --nocapture
```

Or use Neovim's built-in terminal (:terminal) and keep tmux for session persistence only—one full-window nvim is a valid style.

Helix

```
hx .
# use tmux panes for cargo/npm; Helix file picker for buffers
```

Emacs

```
emacs -nw
# or detach-friendly: emacsclient -nw after emacs --daemon
```

Nano / Micro

Fine inside panes; rely on tmux for multi-tasking rather than editor splits.

Comparison: tmux vs screen vs Zellij vs Herdr

Feature	tmux	screen	Zellij	Herdr
Portability	Excellent	Excellent	Good	Growing
Defaults UX	OK	Dated	Great	Mouse + agent UI
Scripting	Excellent	OK	Good	CLI + socket API
Agent awareness	No	No	No	Yes
Typical use	SSH, servers	Legacy	Dev laptops	AI agent herds

Use **tmux** as the baseline skill. Add **Herdr** when agents are the workload.

Troubleshooting

```
# Server alive?
tmux ls

# Stuck session
tmux kill-session -t name

# Reset server (destroys all)
tmux kill-server

# Config error on start
tmux -f /dev/null new -s clean

# 24-bit color test
tmux new 'curl -s https://raw.githubusercontent.com/JohnMorales/dotfiles/master/colors/24-bit'
```

Permission or socket issues after reboot: stale sockets under `/tmp/tmux-$(id -u)/`—remove only when no server should run.

Cheatsheet card

C-b d	detach	C-b c	new window
C-b %	vsplit	C-b "	hsplit
C-b z	zoom	C-b x	kill pane
C-b [	copy mode	C-b]	paste

```
C-b n/p    next/prev window    C-b s      sessions
tmux new -A -s name            attach or create
```

Practice checklist

- ☐ Create, detach, reattach a named session over SSH
- ☐ Build a 3-pane layout and zoom the editor
- ☐ Put vi keys in copy mode and yank a stack trace
- ☐ Write a 15-line script that recreates your dev session
- ☐ Survive an intentional SSH disconnect mid-`sleep 300`

Next: [Herdr](#) for agent-native workspaces, or [Workflows](#) for full recipes with Neovim/Helix/Zed.

Herdr

Herdr

Herdr: Agent-Native Terminal Workspaces

Herdr (pronounced like “herder”) is a terminal workspace manager built for **AI coding agents** and long-lived terminal work. It owns real PTY panes, keeps them running in a background server, and tracks whether each agent is **working**, **blocked**, or **idle**—so you jump to the pane that needs you instead of hunting through tabs.

It is not a replacement for Neovim or Helix. It is the **session layer** those editors (and agent CLIs) live inside—similar in spirit to tmux, but agent-aware.

Official site: <https://herdr.dev/> · Docs: <https://herdr.dev/docs/>

Why Herdr exists

Classic multiplexers (tmux, screen, Zellij) are process-agnostic: a pane is a pane. In 2026 many developers run **Claude Code**, **Codex**, **opencode**, **Cursor agent CLIs**, **Grok**, and similar tools that:

- Run for a long time
- Block waiting for approval or input
- Multiply (one agent per task or worktree)

Herdr adds:

Capability	What you get
Background server	Agents keep running when you detach or close the laptop lid (machine still on)
Agent detection	Sidebar/state for working / blocked / idle
Mouse-first UX	Click panes, drag splits, right-click menus—no prefix required to start
tmux-like prefix	Ctrl-b family bindings if you already know tmux
CLI + socket API	Agents and scripts split panes, wait on status, read output
Remote attach	Work on a box over SSH-style remote modes

Herdr does **not** wrap or replace your agents. You run the same CLIs; Herdr owns their terminals.

Install

Quick install (Linux / macOS)

```
curl -fsSL https://herdr.dev/install.sh | sh
```

Ensure the install location is on your PATH, then:

```
herdr
herdr update          # later, for installer-managed binaries
```

Package managers

```
# Homebrew
brew install herdr

# mise
mise use -g herdr
# fallback if registry is old:
# mise use -g github:herdrdev/herdr

# Nix (pin a release tag in real use)
nix profile install github:herdrdev/herdr/v0.x.y
# or: nix run github:herdrdev/herdr/v0.x.y
```

Manual binary

Download from [GitHub releases](#):

System	Asset
Linux x86_64	herdr-linux-x86_64
Linux aarch64	herdr-linux-aarch64

```
chmod +x herdr-linux-x86_64
mv herdr-linux-x86_64 ~/.local/bin/herdr
```

Channels

Direct installs default to **stable** on Linux/macOS. Preview builds:

```
herdr channel set preview
herdr channel set stable
```

Homebrew / mise / Nix updates go through those tools, not always `herdr update`.

Concepts

```
herdr-server (background)
  session (default or named)
    workspace "api"
      tab "agents"
        pane: claude (blocked)
        pane: codex (working)
      tab "runtime"
        pane: dev server
        pane: nvim / hx
    workspace "docs"
```

Term	Meaning
Server	Owns PTYs; survives client disconnect
Client	Renders UI; attach/detach
Workspace	Project-scale unit (often one repo or concern)
Tab	Layout group inside a workspace
Pane	Real terminal (shell, editor, agent)
Agent state	Rolled up from pane → tab → workspace

Quick start

```
# Start (spawns/connects to server, opens client)
herdr

# Inside: open your usual tools in panes
claude          # or codex, opencode, etc.
nvim
just test

# Detach - leave the herd running (see keyboard docs for exact bind)
# Default family: Ctrl-b then q (detach; leave everything running)
```

Reattach later with `herdr` again. Named sessions:

```
herdr session list
herdr session attach work
herdr session attach side-project
herdr session stop work
herdr session delete side-project
```

Remote-oriented usage (patterns from docs):

```
# SSH to the machine, then run herdr like tmux
ssh you@server
herdr

# Or remote client mode when supported
herdr --remote workbox
herdr --remote workbox --session agents
```

Direct attach by agent label when configured:

```
herdr agent attach reviewer
```

Keyboard and mouse

Herdr is **mouse-native**: click panes, drag borders, use context menus.

If you know tmux, the **prefix** model will feel familiar (**Ctrl-b** by default for many actions). Common ideas from the keyboard guide:

Action	Typical binding
Workspace navigation	prefix+w
Detach (leave running)	prefix+q
New workspace	prefix+Shift+n
Rename workspace	prefix+Shift+w
Close workspace	prefix+Shift+d
Goto picker	prefix+g
Toggle sidebar	prefix+b
Close tab	prefix+Shift+x

Bindings are configurable—see [Configuration](#) for the live reference. Prefer the docs over memorizing a fork’s defaults.

CLI automation

Herdr exposes a CLI (and socket API) so **you or an agent** can drive structure:

```
# Shape a workspace
herdr workspace create --cwd ~/project --label api
herdr tab create --label logs

# Panes
herdr pane split 1-1 --direction right
herdr pane run 1-2 "just test"

# Wait for agent status, then inspect output
herdr wait agent-status 1-1 --status done
herdr pane read 1-2 --source recent-unwrapped
```

Report custom metadata for the UI:

```
herdr pane report-metadata <pane_id> \
  --source my-agent-hook \
  --token model=opus \
  --token summary="reviewing authentication"
```

Exact subcommands evolve—run `herdr --help` and the [CLI / socket API docs](#) for your installed version.

Working with editors

Herdr is the outer frame; pick any editor from this book:

Editor	Pattern
Neovim / LazyVim	Pane for <code>nvim</code> , pane(s) for agents, pane for tests
Helix	Same; Helix picker for files, Herdr for agents
Zed	GUI editor on the host; Herdr for remote/agent PTYs (or local agent panes)
Micro / nano	Fine for quick edits in a pane beside an agent

Example layout intent:

```
Workspace: payments-service

nvim                claude (agent)

                    cargo test -q
```

\$EDITOR still matters

Agents and git often spawn editors:

```
export EDITOR=nvim
export VISUAL=nvim
# or: export EDITOR="zed --wait"
```

Herdr vs tmux vs Zellij

tmux	Zellij	Herdr
PrimaryGeneral sessions job	Friendly multiplexer	Agent herds + sessions
Agent Manual state	Manual	First-class
ScriptingExcellent	Good	CLI + socket API
AvailabilityHighest	Medium	Install binary
on bare VPS		
Mouse Optional UX	Good	Core
Nested Common with tmux	Common	Supported carefully

Practical combo:

- **tmux** on every server as baseline skill
- **Herdr** on machines where you run multiple coding agents
- Avoid deep nesting: do not put *tmux inside Herdr inside tmux* without a reason

Herdr can run **inside** an outer tmux. Conversely, if a shell auto-starts tmux *inside* a Herdr pane, Herdr may see **tmux** as the process instead of the agent—disable auto-tmux in those panes.

Agents and detection

Herdr detects many agent CLIs out of the box (Claude Code, Codex, opencode, Cursor-related tools, Grok, Copilot-style CLIs, and others—see current [agents docs](#)).

States roll up:

- **Blocked** — needs you (approval, question)
- **Working** — busy

- **Idle / done** — quiet

Your job becomes: open the sidebar, jump to **blocked**, answer, move on.

Configuration sketch

Configuration covers keybindings, themes, sidebar, notifications, scrollback, and advanced options. Paths and schema are versioned—start from:

```
# After first run, check docs for config file location and examples
# https://herdr.dev/docs/configuration/
```

When customizing:

1. Change one key family at a time
2. Keep a “panic” path: mouse + command palette / goto picker
3. Store config in git with your dotfiles

Plugins

Herdr supports **plugins** (manifest + executable actions/hooks) and a marketplace direction. Plugins can restore layouts, add overlays, or react to events such as worktree creation.

Sketch (illustrative—verify against current plugin docs):

```
[[actions]]
id = "list-workspaces"
title = "List workspaces"
contexts = ["workspace"]
command = ["node", "index.js"]
```

See [Plugins](#) and [Marketplace](#).

Session state and persistence

Important mental model (aligned with Herdr docs):

- Detach kill: panes keep running in the server
- Restart/restore behavior depends on session state features and agent resume support
- Protocol-breaking upgrades may require stopping the old server (**herdr server stop / herdr session stop <name>**) after **herdr update**

For production data jobs, still use proper supervisors (systemd, containers). Herdr is for **interactive and agent** workspaces, not a substitute for process monitoring of daemons—though many people run dev servers inside panes safely.

Security notes

- Treat a machine running agent CLIs with **repo and cloud credentials** as sensitive
- Remote attach expands the attack surface—use SSH hardening, keys, and least privilege
- Do not paste secrets into agent panes you will share or record
- Review plugin source before installing community plugins

Troubleshooting

Issue	What to try
<code>herdr: command not found</code>	Restart shell; check PATH / installer location
Cannot reattach	<code>herdr session list</code> ; confirm server not stopped
Agent shows as idle/wrong	Pane may be nested tmux/shell wrapper; run agent as direct pane command
After update, client/server mismatch	Stop old server, start fresh client
Keys stolen by outer tmux	Use mouse, or send prefix twice, or simplify nesting

```
herdr update
herdr server stop      # default session server
herdr                  # start clean
```

Practice lab

1. Install Herdr and run `herdr`
2. Create two panes: shell + a long ping or htop
3. Detach, close the terminal emulator, reattach—confirm processes lived
4. Run two agent CLIs (or two shells pretending to be agents) in one workspace
5. Use the sidebar / goto picker to jump between them
6. Script a workspace with `herdr workspace create` and `herdr pane split` if your version supports those verbs

When to use what

```
SSH to random prod bastion, edit nginx.conf
→ tmux + vi/nano
```

```
Local multi-repo app work with one shell
→ tmux or Zellij + nvim/hx/zed
```

```
Three coding agents + dev server + editor
```


→ Herdr (+ your editor of choice)

Further reading

- [Herdr docs home](#)
- [Install](#)
- [Agents](#)
- [How to work](#)
- [Persistence and remote](#)
- [Keyboard](#)
- Agent onboarding prompt from docs: point your agent at `https://herdr.dev/agent-guide.md`

Next: [Zed](#) for a modern native GUI editor, or [Workflows](#) to combine Herdr with Neovim/Helix/tmux deliberately.

Zed

Zed

Zed: Fast Native Code Editor

Zed is a modern, GPU-accelerated code editor written in Rust. It targets snappy local editing, first-class collaboration, and tight language tooling—without Electron. On Linux it is a strong **GUI** counterpart to terminal editors like Neovim and Helix.

Homepage: <https://zed.dev/>

Why Zed on Linux

- **Performance** — native UI, responsive large files and multi-buffer work
- **Language awareness** — tree-sitter + LSP-style intelligence
- **Collaboration** — real-time multiplayer editing (Zed’s signature feature)
- **Vim mode** (optional) — softer landing from modal editors
- **Agent / AI features** — evolving; often used *beside* terminal agents in tmux/Herd
- **Default themes** — **One Dark** / **One Light** (the palette many terminals and this book’s code theme echo)

Zed is **not** a drop-in for rescue-shell editing. Keep **vi**/**nano** skills. Use Zed on workstations and powerful laptops; use TUI editors over raw SSH.

Install

Channels move quickly—prefer the official install docs if a command fails.

```
# Official script (check zed.dev for current recommendation)
curl -f https://zed.dev/install.sh | sh

# Some distros / community packages (availability varies)
# Arch (AUR often): yay -S zed
# Flatpak (if published for your channel): flatpak install ...
# Nix: search nixpkgs for zed-editor
```

Verify:

```
zed --version
which zed
```

Open a project:

```
zed .  
zed path/to/file.rs:42
```

Wait flag for \$EDITOR

GUI editors must block the caller until the file is closed:

```
export EDITOR="zed --wait"  
export VISUAL="$EDITOR"  
git config --global core.editor "zed --wait"
```

Without `--wait`, git commits may open an empty editor and continue immediately.

Core UI map

Area	Role
Buffer	File contents
Tab bar	Open files
Project panel	File tree
Outline / symbols	Jump inside file
Terminal panel	Integrated shell (optional; many still use external tmux)
Command palette	Fuzzy all-the-commands

Command palette is the discovery tool: open it and type what you want (`format`, `theme`, `vim`, `terminal`).

Default chord families (approximate—confirm in Command Palette / docs):

- **Command palette** — often `Ctrl+Shift+P` (Linux)
- **File finder** — often `Ctrl+P`
- **Save** — `Ctrl+S`
- **Toggle terminal** — palette: “terminal”

Keymaps are configurable in Zed settings (JSON).

Settings and config

Zed stores settings in its config directory (typically under `~/.config/zed/` on Linux). Use the UI **Settings** or edit JSON.

Illustrative settings (keys evolve—validate against your version):

```
{
  "theme": "One Dark",
  "buffer_font_family": "JetBrains Mono",
  "buffer_font_size": 14,
  "ui_font_size": 14,
  "tab_size": 2,
  "hard_tabs": false,
  "format_on_save": "on",
  "autosave": "on_focus_change",
  "vim_mode": false,
  "relative_line_numbers": false,
  "soft_wrap": "editor_width",
  "preferred_line_length": 100,
  "remove_trailing_whitespace_on_save": true,
  "ensure_final_newline_on_save": true
}
```

Themes

Built-in **One Dark** and **One Light** match Zed's defaults. Extensions add more. Theme overrides can tweak syntax tokens (see Zed docs on `theme_overrides`).

Vim mode

If you migrate from Vim/Neovim/Helix:

```
{
  "vim_mode": true
}
```

Expect:

- Motions and operators covering daily editing
- Occasional gaps vs full Neovim (plugins, exact text objects)
- Escape to normal mode still fundamental

Helix users: selection-first muscle memory will not map 1:1; use palette and multi-cursor features instead of forcing Helix habits.

Language servers and extensions

Zed uses extensions for languages, themes, and tools.

Typical flow:

1. Open a file type (e.g. `main.go`)
2. Install recommended extension when prompted
3. Confirm diagnostics and go-to-definition work

Install language tools on the host as needed (`rust-analyzer`, `gopls`, `pyright`, etc.). Zed coordinates; it does not always bundle every server.

Projects and workspaces

```
# Prefer opening the project root (git root)
cd ~/src/myapp && zed .

# Multi-folder workspaces: use Zed's workspace UI / recent projects
```

Trust the project panel + file finder over deep nested terminal `cd` when working locally.

Pairing with tmux and Herdr

Zed is a **GUI**. Common architectures:

A. Local Zed + terminal multiplexer for agents

```
Desktop: Zed (code)
Terminal: herdr or tmux
          claude / codex
          just dev
          lazygit
```

B. All in terminal (no Zed remote)

When only SSH is available: use Neovim/Helix inside tmux—not Zed.

C. Remote code

Options people use:

- SSHFS / mutual NFS + local Zed (latency sensitive)
- Dev containers / remote hosts with local sync (mutagen, etc.)
- Stay on terminal editors for high-latency links

Do not expect classic “Zed over dumb SSH” to feel like local.

Collaboration

Zed's multiplayer lets multiple people edit a project together. Use when:

- Pairing on a design spike
- Mentoring
- Shared incident editing (with care around secrets)

Turn collaboration off for sensitive codebases unless policy allows.

Git workflow

Zed provides git UI affordances (status, diffs—feature set depends on version). Many Linux power users still prefer:

```
# In an adjacent terminal / Herdr pane
lazygit
git diff | delta
```

Use Zed for editing hunks; use terminal for complex rebase surgery if you already know it.

Comparison: Zed vs neighbors

	Zed	VSCodium/Code	Neovim	Helix
UI	Native GPU	Electron	TUI	TUI
SSH-first	Weak	Remote-SSH ext	Strong	Strong
Config	JSON / UI	JSON + ext	Lua	TOML
Plugins	Extensions	Huge	Huge	Minimal (built-ins)
Startup	Fast	Heavier	Fast	Fast
Collab	Built-in focus	Live Share etc.	Plugins	Limited

Migration tips

From VS Code

- Install language extensions analogous to your VS Code set
- Relearn a smaller command palette surface
- Map keybindings gradually; do not clone 200 VS Code chords day one

From Neovim

- Enable `vim_mode` or keep a terminal nvim for remote
- Accept that Lua plugin ecosystems do not transfer
- Keep tmux/Herdr muscle memory for session persistence

From Helix

- Use multi-cursor and palette
- Rely on extensions for features Helix builds in

Troubleshooting

Problem	Try
zed not in PATH	Re-login; add install dir to PATH
Git opens and exits	Use <code>zed --wait</code>
Ugly fonts	Set <code>buffer_font_family</code> to a Nerd/ mono font you installed
LSP dead	Install extension + server binary; check language logs in UI
Wayland quirks	Update GPU drivers; try X11 session only for isolation tests
Flatpak permissions	Grant filesystem access to project paths

Practice checklist

- ☐ Install Zed and open a real git repo with `zed .`
- ☐ Set `EDITOR="zed --wait"` and make a git commit
- ☐ Toggle One Dark / One Light
- ☐ Enable or deliberately skip Vim mode
- ☐ Run tests in an external tmux/Herdr pane while editing in Zed
- ☐ Use command palette to format document and jump to file

When to choose Zed

Choose Zed when you want a fast local GUI, collaboration, and modern defaults on Linux desktops.

Choose Neovim/Helix when you live in SSH, minimal environments, or keyboard-only modal purity.

Choose both: Zed locally, terminal editor remotely—the norm for many teams.

Next: [Workflows](#) for end-to-end stacks that combine Zed, tmux, Herdr, and the editors from earlier parts.

Workflows

Workflows

Workflows: Editors + Workspaces + Tools

Isolated chapters teach tools. This chapter **composes** them into daily Linux workflows you can copy and adapt.

Stack templates

1) Server / sysadmin (minimal)

```
SSH → tmux → vi or nano → rg/sed when needed
```

```
ssh -t host 'tmux new -A -s ops'
# inside:
sudoedit /etc/nginx/nginx.conf    # respects $EDITOR if set
journalctl -u nginx -f            # other pane
```

Skills: [Vi](#), [Nano](#), [tmux](#).

2) Terminal IDE (Neovim)

```
tmux session "dev"
  window editor: nvim (LazyVim optional)
  window shell:  tests / services
  window git:    lazygit

export EDITOR=nvim
tmux new -A -s dev -c ~/src/app
# C-b c for new windows; or splits with C-b %
```

Skills: [Neovim](#), [LazyVim](#), [tmux](#), modern tools (lazygit, rg).

3) Selection-first modal (Helix)

```
tmux or bare terminal
hx .
companion pane: cargo test --watch / npm run dev
```

```
export EDITOR=hx
hx .
```

Skills: [Helix](#), [tmux](#).

4) Friendly modeless (Micro)

```
micro config.yaml
# or panes in tmux for logs
```

Skills: [Micro](#).

5) Desktop GUI + agents (Zed + Herdr)

```
Zed → project editing, diagnostics, collab
Herdr → agent CLIs, long commands, detachable state
```

```
export EDITOR="zed --wait"
cd ~/src/app && zed .
# other terminal:
herdr
# panes: claude / codex / just dev
```

Skills: [Zed](#), [Herdr](#).

6) Multi-agent feature work (Herdr-centric)

```
Herdr workspace "feature-x"
  tab agents: agent A (impl) | agent B (tests/review)
  tab runtime: dev server | lazygit | nvim/hx
```

```
herdr
# create panes; run agents; jump to blocked
```

When a pane needs a human editor for a messy conflict:

```
# in a free pane
nvim
# or open GUI if local
zed --wait path/to/conflicted.go
```

7) Emacs as environment

```
emacs -nw or emacsclient -nw
  magit, vterm/eshell, org
optional outer tmux for detach only
```

Skills: [Emacs](#).

Cross-cutting recipes

Open search hits in your editor

```
# ripgrep → fzf → editor
rg -n "TODO|FIXME" | fzf | awk -F: '{print $1, $2}' | while read -r f l; do
  $EDITOR +"$1" "$f"
done

# simpler: fzf file open
export FZF_DEFAULT_COMMAND='fd --type f --hidden --follow --exclude .git'
f() { local f; f=$(fzf --preview 'bat --color=always {}') && $EDITOR "$f"; }
```

Git commit with correct blocking editor

```
# TUI
git config --global core.editor "nvim"
# GUI
git config --global core.editor "zed --wait"
# VSCode
git config --global core.editor "codium --wait"
```

kubectl / sudoedit

```
export EDITOR=nvim
sudoedit /etc/ssh/sshd_config
kubectl edit deploy/api
```

One-shot session script (tmux)

```
#!/usr/bin/env bash
set -euo pipefail
S=app
ROOT=~/.src/app
tmux has-session -t "$S" 2>/dev/null && exec tmux attach -t "$S"
tmux new-session -d -s "$S" -n edit -c "$ROOT"
tmux send-keys -t "$S:edit" 'nvim' Enter
tmux split-window -h -t "$S:edit" -c "$ROOT"
tmux send-keys -t "$S:edit.2" 'git status -sb' Enter
tmux new-window -t "$S" -n run -c "$ROOT"
tmux send-keys -t "$S:run" 'just dev' Enter
tmux attach -t "$S"
```

Agent + editor discipline

1. **One writable branch per agent** (or worktree) to avoid clobbering
2. Human owns merge commits and production credentials
3. Keep secrets out of agent prompts and shared panes
4. Detach Herdr/tmux instead of killing terminals mid-run

```
git worktree add ../app-agent-b -b agent/b
# point second agent cwd at ../app-agent-b
```

Keybinding peace treaty

Tool	Reserve
tmux	Ctrl-b as prefix
Herdr	prefix + mouse; do not nest blindly
Vim/Neovim	leader key (often Space)
Helix	Space for pickers
Terminal emulator	do not bind Ctrl-b / Ctrl-a if you can avoid it

If something “eats” a key, ask: **emulator** → **multiplexer** → **editor** in that order.

Dotfiles layout (suggested)

```
~/.config/  
  nvim/           or helix/   or micro/  
  tmux/tmux.conf  or ~/.tmux.conf  
  git/config  
  zed/settings.json  
  herdr/          (per current Herdr docs)
```

Store in a git repo; bootstrap with a single `install` script. Do not share machine-local secrets in that repo.

Choosing under pressure

```
Production on fire, unfamiliar host  
  → vi or nano, no plugins  
  
Laptop, feature sprint, two agents  
  → Herdr + Zed or nvim  
  
Teaching a newcomer Linux  
  → nano or Micro, then show vi exits  
  
Deep C++/Rust refactor alone  
  → Helix or LazyVim + tmux tests pane
```

Practice: build your “default”

Write this down and stick to it for a month:

```
Daily editor:  _____  
Session tool:  tmux / herdr / both: _____  
$EDITOR:      _____  
Search:       rg + fzf + bat  
Git UI:       lazygit / magit / zed / plain
```

Then deepen *that* stack using the matching chapters—not every chapter at once.

See also

- [Choosing an Editor](#)
- [Terminal Workspaces](#)

- [tmux](#)
- [Herdr](#)
- [Zed](#)
- [Modern Tools](#)

Scripting

Bash

Intro to Shells

Shell vs Terminal: Understanding the Difference

Many people use the terms “shell” and “terminal” interchangeably, but they are actually different components that work together to provide your command-line experience.

What is a Terminal?

A terminal (also called terminal emulator) is a program that provides a window for you to interact with the shell. It handles:

- **Display:** Shows text output from programs
- **Input:** Captures keyboard input and sends it to the shell
- **Window Management:** Provides scrolling, resizing, and visual features
- **Character Encoding:** Handles text rendering and fonts

Common Terminal Emulators

Linux:

- **GNOME Terminal:** Default on GNOME desktop
- **Konsole:** KDE’s terminal emulator
- **xterm:** Classic X11 terminal
- **Alacritty:** GPU-accelerated terminal
- **Terminator:** Terminal with split panes

macOS:

- **Terminal.app:** Built-in terminal
- **iTerm2:** Popular third-party option

Windows:

- **Windows Terminal:** Modern terminal for Windows 10/11
- **Command Prompt:** Traditional Windows terminal
- **PowerShell:** Microsoft’s advanced shell and terminal

What is a Shell?

A shell is the command interpreter that runs inside the terminal. It:

- **Processes Commands:** Interprets what you type
- **Manages Environment:** Handles variables and settings
- **Executes Programs:** Launches other applications
- **Provides Scripting:** Enables automation through scripts

The Relationship

Terminal Emulator

Shell

Your Commands

Practical Examples

Example 1: Multiple Shells in One Terminal

```
# Start in bash
$ echo $SHELL
/bin/bash

# Switch to zsh
$ zsh
$ echo $SHELL
/bin/zsh

# Switch to fish
$ fish
$ echo $SHELL
/usr/bin/fish

# Exit back to previous shells
$ exit # Back to zsh
$ exit # Back to bash
```

Example 2: Same Shell in Different Terminals

You can run the same shell (like bash) in multiple terminal windows simultaneously. Each terminal window is independent, but they're all running the same shell program.

Example 3: Remote Shells

```
# Connect to remote server via SSH
$ ssh user@remote-server
# Now you're running a shell on the remote machine
# but displaying it in your local terminal
```

Terminal Features vs Shell Features

Terminal Features:

- **Colors and Themes:** Visual appearance
- **Font Settings:** Text rendering
- **Window Management:** Tabs, splits, transparency
- **Copy/Paste:** Clipboard integration
- **Scrollback:** History of displayed text

Shell Features:

- **Command History:** Previously executed commands
- **Tab Completion:** Auto-complete functionality
- **Variables:** Environment and user-defined variables
- **Scripting:** Programming capabilities
- **Job Control:** Managing background processes

Configuration Files

Terminal Configuration:

- Usually GUI-based settings
- Profiles for different appearances
- Keyboard shortcuts

Shell Configuration:

```
# Bash configuration files
~/.bashrc      # Interactive non-login shells
~/.bash_profile # Login shells
~/.bash_logout # Logout cleanup

# Zsh configuration files
~/.zshrc       # Main configuration
~/.zprofile    # Login shells

# Fish configuration
~/.config/fish/config.fish
```

Choosing Terminal and Shell Combinations

For Development:

- **Terminal:** iTerm2 (macOS) or GNOME Terminal (Linux)
- **Shell:** bash or zsh with customizations

For System Administration:

- **Terminal:** Any reliable terminal emulator
- **Shell:** bash for compatibility

For Beginners:

- **Terminal:** Default system terminal
- **Shell:** bash (most common) or fish (user-friendly)

Common Misconceptions

Misconception 1: “I’m using Terminal”

Reality: You’re using a terminal emulator to access a shell.

Misconception 2: “Bash is a terminal”

Reality: Bash is a shell that runs inside a terminal.

Misconception 3: “All terminals are the same”

Reality: Different terminals have different features and capabilities.

Practical Tips

Check what terminal you’re using:

```
$ echo $TERM
xterm-256color

$ ps -o comm= -p "$PPID"
gnome-terminal-
```

Check what shell you’re using:

```
$ echo $0
bash

$ ps -p $$
  PID TTY          TIME CMD
 1234 pts/0    00:00:00 bash
```

Launch different shells:

```
# Start a new shell session
$ bash
$ zsh
$ fish

# Run a single command in a different shell
$ zsh -c "echo 'This runs in zsh'"
$ fish -c "echo 'This runs in fish'"
```

Summary

- **Terminal:** The window/interface you see and interact with
- **Shell:** The command interpreter that processes your commands
- **Relationship:** The terminal displays the shell’s input/output
- **Independence:** You can use different shells in the same terminal, or the same shell in different terminals

Understanding this distinction helps you troubleshoot issues, customize your environment effectively, and communicate more precisely about command-line tools.

Types of Shells

Linux and Unix systems support various types of shells, each with its own features and capabilities. Understanding the different shells helps you choose the right tool for your needs.

Common Shell Types

1. Bourne Shell (sh)

- **Original:** The first Unix shell, developed by Stephen Bourne
- **Features:** Basic scripting capabilities, simple syntax
- **Usage:** Still used for system scripts and compatibility
- **Location:** Usually /bin/sh

```
# Check if sh is available
$ which sh
/bin/sh
```

2. Bash (Bourne Again Shell)

- **Most Popular:** Default shell on most Linux distributions
- **Features:** Enhanced version of Bourne shell with many improvements
- **Advantages:**
 - Command history
 - Tab completion
 - Job control
 - Arrays and associative arrays
 - Advanced scripting features

```
# Check Bash version
$ bash --version
GNU bash, version 5.1.16(1)-release
```

3. Zsh (Z Shell)

- **Advanced:** Feature-rich shell with powerful customization
- **Features:**
 - Excellent tab completion
 - Spelling correction
 - Plugin system
 - Themes support
- **Popular Framework:** Oh My Zsh

```
# Install zsh (Ubuntu/Debian)
$ sudo apt install zsh

# Make zsh default shell
$ chsh -s $(which zsh)
```

4. Fish (Friendly Interactive Shell)

- **User-Friendly:** Designed for ease of use
- **Features:**
 - Syntax highlighting
 - Auto-suggestions
 - Web-based configuration
 - No configuration required

```
# Install fish (Ubuntu/Debian)
$ sudo apt install fish

# Start fish shell
$ fish
```

5. Dash (Debian Almquist Shell)

- **Lightweight:** Minimal shell focused on speed
- **Usage:** Often used as `/bin/sh` on Debian-based systems
- **Purpose:** System scripts and boot processes

6. Tcsh (TENEX C Shell)

- **C-like Syntax:** Similar to C programming language
- **Features:** Command history, job control, aliases
- **Usage:** Less common in modern systems

7. Ksh (Korn Shell)

- **Commercial:** Developed by David Korn at Bell Labs
- **Features:** Combines features of Bourne and C shells
- **Variants:** ksh88, ksh93, mksh (MirBSD Korn Shell)

Checking Available Shells

View all available shells:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/zsh
/usr/bin/fish
```

Check your current shell:

```
$ echo $SHELL
/bin/bash

# Or use ps command
$ ps -p $$
  PID TTY          TIME CMD
 1234 pts/0    00:00:00 bash
```

Check shell being used by a script:

```
$ ps -o pid,ppid,cmd
```

Shell Comparison

Feature	sh	bash	zsh	fish	dash
Speed	Fast	Medium	Medium	Medium	Very Fast
Scripting	Basic	Advanced	Advanced	Good	Basic

Feature	sh	bash	zsh	fish	dash
Interactive	Basic	Good	Excellent	Excellent	Basic
Compatibility	High	High	Good	Low	High
Customization	Limited	Good	Excellent	Good	Limited

Choosing the Right Shell

For Scripting:

- **bash**: Best balance of features and compatibility
- **sh**: Maximum portability across systems
- **dash**: When speed is critical

For Interactive Use:

- **zsh**: Power users who want customization
- **fish**: Beginners who want user-friendly features
- **bash**: Good default choice for most users

For System Administration:

- **bash**: Most common, good documentation
- **sh**: For system scripts that need portability

Switching Between Shells

Temporarily switch:

```
# Start a new shell session
$ zsh
$ fish
$ bash

# Exit back to previous shell
$ exit
```

Change default shell:

```
# Change default shell for current user
$ chsh -s /usr/bin/zsh

# Verify the change
$ grep $USER /etc/passwd
```

Shell Compatibility

When writing scripts, consider compatibility:

```
#!/bin/bash
# This script requires bash-specific features

#!/bin/sh
# This script should work with any POSIX shell

#!/usr/bin/env bash
# Portable way to find bash
```

Understanding the different types of shells available helps you make informed decisions about which shell to use for different tasks and environments.

What is a Shell?

A shell is a command-line interface that acts as an intermediary between you and the operating system kernel. It's called a "shell" because it wraps around the kernel, providing a user-friendly way to interact with the system.

The Role of a Shell

The shell serves several critical functions:

1. **Command Interpreter:** Reads and executes commands you type
2. **Program Launcher:** Starts other programs and applications
3. **Environment Manager:** Manages environment variables and settings
4. **Scripting Platform:** Allows automation through script files

How Shells Work

When you type a command, here's what happens:

```
$ ls -la
```

1. The shell reads your input
2. Parses the command and arguments
3. Locates the program (`ls`) in the system PATH
4. Executes the program with the specified arguments (`-la`)
5. Displays the output back to you

Interactive vs Non-Interactive Shells

Interactive Shell

- Accepts commands from user input
- Provides a prompt (like `$` or `#`)
- Offers features like command history and tab completion

Non-Interactive Shell

- Executes scripts without user interaction
- Used for automation and batch processing
- No prompt or interactive features

Login vs Non-Login Shells

Login Shell

- Started when you log into the system
- Reads configuration files like `.bash_profile`
- Sets up the initial environment

Non-Login Shell

- Started from within another shell or application
- Reads different configuration files like `.bashrc`
- Inherits environment from parent shell

Shell Capabilities

Modern shells provide powerful features:

- **Command History:** Access previously executed commands
- **Tab Completion:** Auto-complete commands and filenames
- **Wildcards:** Use patterns like `*.txt` to match multiple files
- **Pipes and Redirection:** Chain commands and redirect output
- **Variables:** Store and manipulate data
- **Control Structures:** Conditional statements and loops
- **Functions:** Create reusable code blocks

Example: Basic Shell Interaction

```
# Display current directory
$ pwd
/home/username

# List files
$ ls
Documents  Downloads  Pictures
```

```
# Create a directory
$ mkdir my_scripts

# Navigate to the directory
$ cd my_scripts

# Check current location
$ pwd
/home/username/my_scripts
```

Understanding what a shell is and how it works is fundamental to becoming proficient in shell scripting and system administration.

Scripting

Your First Shell Script

Let's create your first shell script step by step. This hands-on approach will help you understand the basics and get you started with shell scripting.

Step 1: Create Your First Script

Using a Text Editor

Open your favorite text editor and create a new file called `hello.sh`:

```
# Using nano
$ nano hello.sh

# Using vim
$ vim hello.sh

# Using VS Code
$ code hello.sh
```

Write the Script

Type the following content into your file:

```
#!/bin/bash
# My first shell script
# Purpose: Display a greeting message

echo "Hello, World!"
echo "Welcome to shell scripting!"
echo "Today's date is: $(date)"
echo "Current user: $(whoami)"
echo "Current directory: $(pwd)"
```

Save the File

- **nano:** Press `Ctrl+X`, then `Y`, then `Enter`
- **vim:** Press `Esc`, type `:wq`, then `Enter`
- **VS Code:** Press `Ctrl+S`

Step 2: Understanding the Script

Let's break down each part:

Shebang Line

```
#!/bin/bash
```

- **Purpose:** Tells the system to use bash to execute this script
- **Must be:** The very first line of the script
- **Alternative:** `#!/bin/sh` for POSIX compatibility

Comments

```
# My first shell script  
# Purpose: Display a greeting message
```

- **Purpose:** Document what the script does
- **Syntax:** Lines starting with `#` (except shebang)
- **Best Practice:** Always comment your scripts

Commands

```
echo "Hello, World!"
```

- **echo:** Command to display text
- **Quotes:** Protect text with spaces and special characters

Command Substitution

```
echo "Today's date is: $(date)"
```

- **\$(command):** Executes command and substitutes its output
- **Alternative:** Backticks ``date`` (older syntax)

Step 3: Make the Script Executable

Before you can run the script, you need to make it executable:

```
# Check current permissions  
$ ls -l hello.sh
```

```
-rw-r--r-- 1 user user 245 Jan 15 10:30 hello.sh

# Make it executable
$ chmod +x hello.sh

# Verify permissions changed
$ ls -l hello.sh
-rwxr-xr-x 1 user user 245 Jan 15 10:30 hello.sh
```

Understanding Permissions

- **r**: Read permission
- **w**: Write permission
- **x**: Execute permission
- **First rwx**: Owner permissions
- **Second rwx**: Group permissions
- **Third rwx**: Other users permissions

Step 4: Run Your Script

There are several ways to execute your script:

Method 1: Direct Execution

```
$ ./hello.sh
Hello, World!
Welcome to shell scripting!
Today's date is: Mon Jan 15 10:35:22 EST 2024
Current user: john
Current directory: /home/john/scripts
```

Method 2: Using bash Command

```
$ bash hello.sh
# Same output as above
```

Method 3: Using sh Command

```
$ sh hello.sh
# Same output as above
```

Method 4: Full Path

```
$ /home/john/scripts/hello.sh
# Same output as above
```

Step 5: Enhance Your Script

Let's make the script more interactive:

```
#!/bin/bash
# Enhanced greeting script
# Purpose: Interactive greeting with user input

echo "=====
echo "    Welcome to Shell Scripting!"
echo "=====
echo

# Get user's name
echo "What's your name?"
read name

# Get user's favorite color
echo "What's your favorite color?"
read color

# Display personalized greeting
echo
echo "Hello, $name!"
echo "I see your favorite color is $color."
echo "That's a great choice!"
echo
echo "System Information:"
echo "=====
echo "Date: $(date)"
echo "User: $(whoami)"
echo "Home Directory: $HOME"
echo "Current Directory: $(pwd)"
```



```
echo "Shell: $SHELL"
echo
echo "Have a great day, $name!"
```

Save this as `enhanced_hello.sh` and run it:

```
$ chmod +x enhanced_hello.sh
$ ./enhanced_hello.sh
```

Step 6: Adding Error Handling

Let's add some basic error handling:

```
#!/bin/bash
# Script with error handling
# Purpose: Demonstrate basic error handling

set -e # Exit immediately if a command exits with a non-zero status

echo "Starting script..."

# Check if a directory exists
if [ -d "/home" ]; then
    echo " /home directory exists"
else
    echo " /home directory not found"
    exit 1
fi

# Try to create a test file
TEST_FILE="test_file.txt"
echo "Creating test file: $TEST_FILE"

if echo "This is a test" > "$TEST_FILE"; then
    echo " Test file created successfully"

    # Clean up
    rm "$TEST_FILE"
    echo " Test file cleaned up"
else
    echo " Failed to create test file"
    exit 1
fi

echo "Script completed successfully!"
```

Step 7: Script with Functions

Let's organize code using functions:

```
#!/bin/bash
# Script with functions
# Purpose: Demonstrate function usage

# Function to display a separator
show_separator() {
    echo "===== "
}

# Function to display system info
show_system_info() {
    echo "System Information:"
    echo "OS: $(uname -s)"
    echo "Kernel: $(uname -r)"
    echo "Architecture: $(uname -m)"
    echo "Hostname: $(hostname)"
}

# Function to display disk usage
show_disk_usage() {
    echo "Disk Usage:"
    df -h | head -5
}

# Main script execution
main() {
    show_separator
    echo "    System Status Report"
    show_separator
    echo

    show_system_info
    echo

    show_disk_usage
    echo

    show_separator
    echo "Report generated on: $(date)"
    show_separator
}

# Call main function
```

```
main
```

Common Beginner Mistakes

1. Forgetting the Shebang

```
# Wrong - no shebang
echo "Hello World"

# Correct
#!/bin/bash
echo "Hello World"
```

2. Not Making Script Executable

```
# This will fail
$ ./myscript.sh
bash: ./myscript.sh: Permission denied

# Fix it
$ chmod +x myscript.sh
$ ./myscript.sh
```

3. Wrong Path in Shebang

```
# Might not work on all systems
#!/usr/bin/bash

# More portable
#!/bin/bash

# Most portable
#!/usr/bin/env bash
```

4. Spaces in Variable Assignment

```
# Wrong - spaces around =
name = "John"

# Correct - no spaces
```

```
name="John"
```

Best Practices for Beginners

1. Always Use Shebang

```
#!/bin/bash
```

2. Add Comments

```
# What the script does  
# Author and date  
# Usage instructions
```

3. Use Meaningful Names

```
# Good  
backup_directory="/backup"  
user_name="john"  
  
# Avoid  
dir="/backup"  
n="john"
```

4. Quote Variables

```
# Good  
echo "Hello, $name"  
cp "$source_file" "$destination"  
  
# Risky  
echo Hello, $name  
cp $source_file $destination
```

5. Check for Errors

```
#!/bin/bash
set -e # Exit on error

# Or check individual commands
if ! command_that_might_fail; then
    echo "Command failed!"
    exit 1
fi
```

Testing Your Scripts

1. Syntax Check

```
$ bash -n myscript.sh
# No output means no syntax errors
```

2. Debug Mode

```
$ bash -x myscript.sh
# Shows each command as it's executed
```

3. Use shellcheck

```
$ shellcheck myscript.sh
# Provides suggestions for improvement
```

Next Steps

Now that you've created your first script, you can:

1. **Experiment:** Modify the examples and see what happens
2. **Practice:** Create scripts for tasks you do regularly
3. **Learn More:** Study variables, conditionals, and loops
4. **Read Others' Scripts:** Look at system scripts in `/etc/init.d/`
5. **Use Version Control:** Start tracking your scripts with git

Congratulations! You've written and executed your first shell script. This is the foundation for all the advanced scripting techniques you'll learn in the following chapters.

Script Execution Methods

Understanding how to execute shell scripts is crucial for effective shell scripting. There are multiple ways to run scripts, each with its own use cases and implications.

Execution Methods Overview

Method	Syntax	New Process	Environment	Use Case
Direct	<code>./script.sh</code>	Yes	Inherited	Most common
Interpreter	<code>bash script.sh</code>	Yes	Inherited	Testing/debugging
Source	<code>source script.sh</code>	No	Modified	Environment setup
Absolute Path	<code>/path/to/script.sh</code>	Yes	Inherited	System scripts

Method 1: Direct Execution

This is the most common way to run scripts.

Prerequisites

```
# Script must be executable
$ chmod +x myscript.sh

# Script must have proper shebang
$ head -1 myscript.sh
#!/bin/bash
```

Execution

```
# Run from current directory
$ ./myscript.sh

# Run from anywhere (if in PATH)
```

```
$ myscript.sh
```

Example Script

```
#!/bin/bash
# direct_example.sh
echo "Running in process: $$"
echo "Parent process: $PPID"
echo "Current directory: $(pwd)"
export NEW_VAR="Hello from script"
```

Test It

```
$ chmod +x direct_example.sh
$ ./direct_example.sh
Running in process: 12345
Parent process: 12344
Current directory: /home/user/scripts

# Check if NEW_VAR exists in current shell
$ echo $NEW_VAR
# (empty - variable doesn't exist in parent shell)
```

Method 2: Using Shell Interpreter

Run the script by explicitly calling the shell interpreter.

Bash Interpreter

```
$ bash myscript.sh
```

Other Interpreters

```
$ sh myscript.sh      # POSIX shell
$ zsh myscript.sh     # Z shell
$ dash myscript.sh    # Debian Almquist shell
```

Advantages

- **No execute permission needed**
- **Override shebang:** Force specific interpreter
- **Testing:** Test script with different shells

Example

```
# Create script without execute permission
$ cat > test_interpreter.sh << 'EOF'
#!/bin/bash
echo "Shell: $0"
echo "Bash version: $BASH_VERSION"
EOF

# Run without making executable
$ bash test_interpreter.sh
Shell: bash
Bash version: 5.1.16(1)-release

# Try with different shell
$ sh test_interpreter.sh
Shell: sh
Bash version:
```

Method 3: Source (Dot) Command

Execute script in the current shell environment.

Syntax

```
$ source myscript.sh
# or
$ . myscript.sh
```

Key Characteristics

- **No new process:** Runs in current shell
- **Environment changes persist:** Variables and functions remain
- **No shebang needed:** Uses current shell
- **No execute permission needed**

Example Script

```
# source_example.sh
echo "Setting up environment..."
export PROJECT_ROOT="/home/user/myproject"
export PATH="$PROJECT_ROOT/bin:$PATH"

# Define a function
greet() {
    echo "Hello from sourced function!"
}

echo "Environment setup complete"
echo "PROJECT_ROOT: $PROJECT_ROOT"
```

Test Sourcing

```
# Before sourcing
$ echo $PROJECT_ROOT
# (empty)

# Source the script
$ source source_example.sh
Setting up environment...
Environment setup complete
PROJECT_ROOT: /home/user/myproject

# After sourcing - variables persist
$ echo $PROJECT_ROOT
/home/user/myproject

# Function is available
$ greet
Hello from sourced function!
```

Method 4: Absolute Path Execution

Run script using its full path.

Examples

```
# Full path execution
$ /home/user/scripts/myscript.sh

# Using $HOME variable
$ $HOME/scripts/myscript.sh

# Using tilde expansion
$ ~/scripts/myscript.sh
```

Use Cases

- **System scripts:** /etc/init.d/apache2 start
- **Cron jobs:** Full paths prevent PATH issues
- **Remote execution:** SSH with full paths

Method 5: Command Substitution

Execute script and capture output.

Syntax

```
# Modern syntax
result=$(./myscript.sh)

# Legacy syntax (avoid)
result=`./myscript.sh`
```

Example

```
# Script that returns a value
$ cat > get_timestamp.sh << 'EOF'
#!/bin/bash
date +%Y%m%d_%H%M%S
EOF

$ chmod +x get_timestamp.sh

# Capture output
$ timestamp=$(./get_timestamp.sh)
```

```
$ echo "Backup file: backup_${timestamp}.tar.gz"
Backup file: backup_20240115_143022.tar.gz
```

Execution Context and Environment

Process Creation

```
#!/bin/bash
# process_info.sh
echo "Script PID: $$"
echo "Parent PID: $PPID"
echo "Shell: $0"
ps -f --pid $$ --pid $PPID
```

Environment Inheritance

```
#!/bin/bash
# env_test.sh
echo "PATH: $PATH"
echo "HOME: $HOME"
echo "USER: $USER"
echo "Custom var: $MY_CUSTOM_VAR"
```

Test Environment

```
# Set a custom variable
$ export MY_CUSTOM_VAR="test value"

# Run script - inherits environment
$ ./env_test.sh
PATH: /usr/local/bin:/usr/bin:/bin
HOME: /home/user
USER: user
Custom var: test value
```

Script Arguments and Parameters

Passing Arguments

```
$ ./myscript.sh arg1 arg2 arg3
```

Accessing Arguments

```
#!/bin/bash
# args_example.sh
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Test Arguments

```
$ ./args_example.sh hello world 123
Script name: ./args_example.sh
First argument: hello
Second argument: world
All arguments: hello world 123
Number of arguments: 3
```

Exit Status and Error Handling

Exit Status

```
#!/bin/bash
# exit_status.sh
echo "Performing operation..."

# Simulate success or failure
if [ "$1" = "fail" ]; then
    echo "Operation failed!"
    exit 1
else
    echo "Operation successful!"
    exit 0
fi
```

Check Exit Status

```
$ ./exit_status.sh success
Operation successful!
$ echo $?
0

$ ./exit_status.sh fail
Operation failed!
$ echo $?
1
```

Advanced Execution Techniques

Background Execution

```
# Run script in background
$ ./long_running_script.sh &
[1] 12345

# Check background jobs
$ jobs
[1]+  Running    ./long_running_script.sh &
```

Conditional Execution

```
# Run second command only if first succeeds
$ ./script1.sh && ./script2.sh

# Run second command only if first fails
$ ./script1.sh || ./script2.sh

# Always run second command
$ ./script1.sh; ./script2.sh
```

Piping Script Output

```
# Pipe script output to another command
$ ./generate_data.sh | sort | uniq

# Redirect output to file
$ ./report_script.sh > report.txt 2>&1
```

Debugging Script Execution

Debug Mode

```
# Show commands as they execute
$ bash -x myscript.sh

# Or add to script
#!/bin/bash -x
```

Verbose Mode

```
# Show input lines as they're read
$ bash -v myscript.sh
```

Syntax Check

```
# Check syntax without executing
$ bash -n myscript.sh
```

Combined Options

```
# Multiple debug options
$ bash -xv myscript.sh
```

Security Considerations

File Permissions

```
# Secure script permissions
$ chmod 755 myscript.sh # Owner: rwx, Others: r-x
$ chmod 700 myscript.sh # Owner only: rwx
```

PATH Security

```
#!/bin/bash
# Use absolute paths for security
/bin/ls /home/user
/usr/bin/whoami
```

Input Validation

```
#!/bin/bash
# Validate input before execution
if [ $# -ne 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

filename="$1"
if [ ! -f "$filename" ]; then
    echo "Error: File '$filename' not found"
    exit 1
fi
```

Best Practices

1. Always Use Shebang

```
#!/bin/bash
# or
#!/usr/bin/env bash # More portable
```

2. Set Error Handling

```
#!/bin/bash
set -euo pipefail # Exit on error, undefined vars, pipe failures
```

3. Use Appropriate Execution Method

- **Direct execution:** General purpose scripts
- **Source:** Environment setup scripts
- **Interpreter:** Testing and debugging
- **Background:** Long-running processes

4. Handle Arguments Properly

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <arguments>"
    exit 1
fi
```

5. Provide Clear Output

```
#!/bin/bash
echo "Starting process..."
# ... script logic ...
echo "Process completed successfully"
```

Understanding these execution methods gives you the flexibility to run scripts in the most appropriate way for your specific needs and environment.

What is Shell Scripting?

Shell scripting is the practice of writing programs (scripts) that are executed by a shell. These scripts automate tasks, combine multiple commands, and create powerful workflows that would be tedious to perform manually.

Definition and Purpose

A shell script is a text file containing a series of commands that the shell can execute. Instead of typing commands one by one, you write them in a file and execute the entire sequence at once.

Key Benefits:

- **Automation:** Eliminate repetitive manual tasks
- **Consistency:** Ensure tasks are performed the same way every time
- **Efficiency:** Save time and reduce human error
- **Scheduling:** Run scripts automatically at specific times
- **Complex Logic:** Implement decision-making and loops

When to Use Shell Scripts

Perfect for:

- **System Administration:** User management, backups, log rotation
- **File Operations:** Batch processing, file organization
- **Development Workflows:** Build processes, deployment scripts
- **Data Processing:** Log analysis, report generation
- **System Monitoring:** Health checks, alert systems

Not Ideal for:

- **Complex Calculations:** Use Python, R, or specialized tools
- **GUI Applications:** Shell scripts are command-line oriented
- **Performance-Critical Tasks:** Compiled languages are faster
- **Cross-Platform Applications:** Shell scripts are OS-specific

Types of Shell Scripts

1. Simple Command Sequences

```
#!/bin/bash
# Backup script
cp /home/user/documents/* /backup/
tar -czf /backup/backup_$(date +%Y%m%d).tar.gz /backup/*.txt
echo "Backup completed at $(date)"
```

2. Interactive Scripts

```
#!/bin/bash
# User input script
echo "What's your name?"
read name
echo "Hello, $name! Welcome to shell scripting."
```

3. System Administration Scripts

```
#!/bin/bash
# System monitoring script
echo "System Status Report - $(date)"
echo "======"
echo "Disk Usage:"
df -h
echo "Memory Usage:"
free -h
echo "CPU Load:"
uptime
```

4. Data Processing Scripts

```
#!/bin/bash
# Log analysis script
echo "Analyzing web server logs..."
grep "ERROR" /var/log/apache2/error.log | wc -l
echo "Total errors found"
```

Shell Script Components

1. Shebang Line

```
#!/bin/bash
# Tells the system which interpreter to use
```

2. Comments

```
# This is a single-line comment
: '
This is a
multi-line comment
'
```

3. Variables

```
name="John"
age=25
echo "Name: $name, Age: $age"
```

4. Commands

```
ls -la
pwd
date
```

5. Control Structures

```
if [ $age -gt 18 ]; then
    echo "Adult"
else
    echo "Minor"
fi
```

Script Execution Methods

1. Make Executable and Run

```
# Make script executable
$ chmod +x myscript.sh

# Run the script
$ ./myscript.sh
```

2. Run with Shell Interpreter

```
# Run with bash
$ bash myscript.sh

# Run with sh
$ sh myscript.sh
```

3. Source the Script

```
# Execute in current shell (affects current environment)
$ source myscript.sh
# or
$ . myscript.sh
```

Script Structure Best Practices

Basic Template:

```
#!/bin/bash

# Script: example.sh
# Purpose: Demonstrate shell script structure
# Author: Your Name
# Date: $(date)

# Set strict error handling
set -euo pipefail

# Global variables
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
```

```

LOG_FILE="/var/log/myscript.log"

# Functions
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a "$LOG_FILE"
}

main() {
    log_message "Script started"

    # Main script logic here
    echo "Hello, World!"

    log_message "Script completed"
}

# Execute main function
main "$@"

```

Common Shell Scripting Patterns

1. Error Handling

```

#!/bin/bash
set -e # Exit on any error

command_that_might_fail || {
    echo "Command failed!"
    exit 1
}

```

2. Argument Processing

```

#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

filename="$1"
echo "Processing file: $filename"

```

3. Configuration Files

```
#!/bin/bash
# Load configuration
CONFIG_FILE="config.conf"
if [ -f "$CONFIG_FILE" ]; then
    source "$CONFIG_FILE"
else
    echo "Configuration file not found!"
    exit 1
fi
```

Shell Scripting vs Other Languages

Shell Scripts Excel At:

- System integration
- File manipulation
- Process management
- Quick automation tasks
- Gluing together existing tools

Other Languages Better For:

- **Python:** Complex logic, data analysis, web development
- **C/C++:** Performance-critical applications
- **JavaScript:** Web applications, Node.js servers
- **Go:** Network services, system tools

Development Environment

Essential Tools:

- **Text Editor:** vim, nano, VS Code, or any editor
- **Shell:** bash, zsh, or your preferred shell
- **Version Control:** git for tracking changes
- **Testing:** shellcheck for syntax checking

Useful Commands:

```
# Check script syntax
$ bash -n myscript.sh

# Debug script execution
$ bash -x myscript.sh

# Use shellcheck for best practices
$ shellcheck myscript.sh
```

Real-World Example

Here's a practical backup script:

```
#!/bin/bash
# backup.sh - Simple backup script

# Configuration
SOURCE_DIR="/home/user/documents"
BACKUP_DIR="/backup"
DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_NAME="backup_$(date +%Y%m%d_%H%M%S).tar.gz"

# Create backup directory if it doesn't exist
mkdir -p "$BACKUP_DIR"

# Create backup
echo "Starting backup of $SOURCE_DIR..."
tar -czf "$BACKUP_DIR/$BACKUP_NAME" "$SOURCE_DIR"

# Check if backup was successful
if [ $? -eq 0 ]; then
    echo "Backup completed successfully: $BACKUP_DIR/$BACKUP_NAME"

    # Remove backups older than 7 days
    find "$BACKUP_DIR" -name "backup_*.tar.gz" -mtime +7 -delete
    echo "Old backups cleaned up"
else
    echo "Backup failed!"
    exit 1
fi
```

Shell scripting is a powerful skill that bridges the gap between simple command-line usage and full programming. It's an essential tool for anyone working with Linux or Unix systems.

Bash Features

Bash Environment

The Bash environment consists of variables, settings, and configurations that determine how Bash behaves. Understanding and managing this environment is crucial for effective shell scripting and system administration.

Environment Variables

Environment variables are key-value pairs that affect how programs run. They're inherited by child processes and available to all programs launched from the shell.

Viewing Environment Variables

```
# Display all environment variables
$ env
PATH=/usr/local/bin:/usr/bin:/bin
HOME=/home/username
USER=username
SHELL=/bin/bash

# Display specific variable
$ echo $HOME
/home/username

# Alternative method
$ printenv HOME
/home/username

# List all variables (including shell variables)
$ set
```

Common Environment Variables

System Variables

```
# User information
echo $USER          # Current username
echo $HOME          # User's home directory
echo $UID           # User ID number
echo $GROUPS        # User's group memberships

# System information
echo $HOSTNAME      # System hostname
echo $HOSTTYPE      # Machine type
echo $OSTYPE        # Operating system type
echo $MACHINE_TYPE  # Machine description

# Shell information
echo $SHELL         # Path to current shell
echo $BASH_VERSION  # Bash version
echo $0             # Name of current script/shell
```

Path Variables

```
# Executable search path
echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin

# Library search path
echo $LD_LIBRARY_PATH

# Manual page search path
echo $MANPATH
```

Locale Variables

```
# Language and locale settings
echo $LANG          # Default locale
echo $LC_ALL        # Override all locale settings
echo $LC_TIME       # Time format locale
echo $LC_NUMERIC    # Number format locale
```

Setting Environment Variables

Temporary Variables (Current Session)

```
# Set variable for current session
$ export MY_VAR="Hello World"
$ echo $MY_VAR
Hello World

# Alternative syntax
$ MY_VAR="Hello World"
$ export MY_VAR
```

Permanent Variables

```
# Add to ~/.bashrc for user-specific variables
echo 'export MY_APP_PATH="/opt/myapp"' >> ~/.bashrc

# Add to /etc/environment for system-wide variables (Ubuntu/Debian)
echo 'MY_GLOBAL_VAR="value"' | sudo tee -a /etc/environment

# Add to /etc/profile for system-wide variables (all systems)
echo 'export GLOBAL_PATH="/usr/local/custom"' | sudo tee -a /etc/profile
```

Unsetting Variables

```
# Remove environment variable
$ unset MY_VAR
$ echo $MY_VAR
# (empty output)

# Remove from environment but keep as shell variable
$ export -n MY_VAR
```

Shell Variables vs Environment Variables

Shell Variables

Only available in the current shell:

```
# Shell variable (not exported)
$ my_shell_var="local value"
$ echo $my_shell_var
local value

# Start subshell - variable not available
$ bash
$ echo $my_shell_var
# (empty)
$ exit
```

Environment Variables

Available to child processes:

```
# Environment variable (exported)
$ export my_env_var="global value"
$ echo $my_env_var
global value

# Start subshell - variable is available
$ bash
$ echo $my_env_var
global value
$ exit
```

Special Variables

Bash provides several special variables with predefined meanings:

Positional Parameters

```
#!/bin/bash
# script_args.sh
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

```
echo "All arguments as single string: $*"

```

Process Variables

```
#!/bin/bash
echo "Current process ID: $$"
echo "Parent process ID: $PPID"
echo "Last background process ID: $!"
echo "Exit status of last command: $?"
```

Test Special Variables

```
$ ./script_args.sh hello world test
Script name: ./script_args.sh
First argument: hello
Second argument: world
All arguments: hello world test
Number of arguments: 3
All arguments as single string: hello world test
```

Configuration Files

Bash reads various configuration files at startup, depending on how it's invoked:

Login Shell Files (in order)

1. /etc/profile - System-wide login settings
2. ~/.bash_profile - User login settings
3. ~/.bash_login - Alternative user login settings
4. ~/.profile - POSIX-compatible user settings

Non-Login Interactive Shell Files

1. /etc/bash.bashrc - System-wide interactive settings
2. ~/.bashrc - User interactive settings

Non-Interactive Shell Files

- Uses \$BASH_ENV if set

Logout Files

- ~/.bash_logout - Executed when login shell exits

Configuration File Examples

~/.bashrc

```
# ~/.bashrc - User interactive shell configuration

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
alias ll='ls -la'
alias la='ls -A'
alias l='ls -CF'

# Custom prompt
PS1='\u@\h:\w\$ '

# Environment variables
export EDITOR=vim
export PAGER=less

# Add personal bin to PATH
if [ -d "$HOME/bin" ]; then
    PATH="$HOME/bin:$PATH"
fi

# Custom functions
mkcd() {
    mkdir -p "$1" && cd "$1"
}

# History settings
HISTSIZE=1000
HISTFILESIZE=2000
HISTCONTROL=ignoredups
```

~/.bash_profile

```
# ~/.bash_profile - User login shell configuration

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
```

```

fi

# User specific environment and startup programs
export PATH="$HOME/.local/bin:$PATH"

# Start SSH agent
if [ -z "$SSH_AUTH_SOCK" ]; then
    eval $(ssh-agent -s)
fi

# Welcome message
echo "Welcome, $USER!"
echo "Today is $(date)"

```

Prompt Customization

The shell prompt is controlled by the PS1 variable:

Basic Prompt Variables

```

# Default prompt
PS1='\u@\h:\w\$ '
# Results in: user@hostname:/current/path$

# Prompt escape sequences
\u # Username
\h # Hostname (short)
\H # Hostname (full)
\w # Current working directory
\W # Basename of current directory
\d # Date
\t # Time (24-hour)
\T # Time (12-hour)
\$ # $ for regular user, # for root

```

Colorized Prompt

```

# Colors
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'

```

```
NC='\033[0m' # No Color

# Colorized prompt
PS1='${GREEN}\u@\h${NC}:${BLUE}\w${NC}\$ ' "
```

Advanced Prompt with Git Status

```
# Function to show git branch
git_branch() {
    git branch 2>/dev/null | grep '^*' | colrm 1 2
}

# Prompt with git branch
PS1='\u@\h:\w$(git_branch)\$ ' "
```

PATH Management

The PATH variable determines where the shell looks for executable files:

Viewing PATH

```
# Display PATH
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin

# Display PATH in readable format
$ echo $PATH | tr ':' '\n'
/usr/local/bin
/usr/bin
/bin
/usr/local/sbin
/usr/sbin
/sbin
```

Modifying PATH

```
# Add directory to beginning of PATH
export PATH="/new/directory:$PATH"

# Add directory to end of PATH
export PATH="$PATH:/new/directory"
```



```
# Add multiple directories
export PATH="/dir1:/dir2:$PATH:/dir3"
```

PATH Best Practices

```
# Check if directory exists before adding
if [ -d "$HOME/bin" ]; then
    export PATH="$HOME/bin:$PATH"
fi

# Remove duplicate entries
PATH=$(echo "$PATH" | awk -v RS=':' -v ORS=":" '!a[$1]++{if (NR > 1) printf ORS; printf $a[$1]}'
```

Shell Options and Settings

Set Options

```
# Display all set options
$ set -o

# Enable options
set -o errexit      # Exit on error
set -o nounset      # Exit on undefined variable
set -o pipefail     # Exit on pipe failure

# Disable options
set +o errexit      # Disable exit on error

# Short form
set -e              # Same as set -o errexit
set -u              # Same as set -o nounset
set -x              # Enable debug mode
```

Shopt Options

```
# Display all shopt options
$ shopt

# Enable options
shopt -s extglob     # Extended globbing
shopt -s globstar    # Recursive globbing (**)
```

```
shopt -s nocaseglob # Case-insensitive globbing

# Disable options
shopt -u extglob     # Disable extended globbing
```

Environment Management Scripts

Environment Setup Script

```
#!/bin/bash
# setup_env.sh - Development environment setup

echo "Setting up development environment..."

# Java environment
export JAVA_HOME="/usr/lib/jvm/java-11-openjdk"
export PATH="$JAVA_HOME/bin:$PATH"

# Node.js environment
export NODE_PATH="/usr/local/lib/node_modules"
export PATH="/usr/local/node/bin:$PATH"

# Python environment
export PYTHONPATH="/usr/local/lib/python3.9/site-packages:$PYTHONPATH"

# Custom application paths
export APP_HOME="/opt/myapp"
export PATH="$APP_HOME/bin:$PATH"

# Development tools
export EDITOR=vim
export BROWSER=firefox

echo "Environment setup complete!"
echo "Java version: $(java -version 2>&1 | head -1)"
echo "Node version: $(node --version 2>/dev/null || echo 'Not installed')"
echo "Python version: $(python3 --version)"
```

Environment Backup and Restore

```
#!/bin/bash
# env_backup.sh - Backup current environment

BACKUP_FILE="env_backup_$(date +%Y%m%d_%H%M%S).sh"

echo "#!/bin/bash" > "$BACKUP_FILE"
echo "# Environment backup created on $(date)" >> "$BACKUP_FILE"
echo "" >> "$BACKUP_FILE"

# Backup important variables
for var in PATH HOME USER SHELL LANG; do
    echo "export $var='${!var}'" >> "$BACKUP_FILE"
done

echo "Environment backed up to $BACKUP_FILE"
```

Debugging Environment Issues

Common Environment Problems

```
# Check if command is found
$ which python3
/usr/bin/python3

$ type python3
python3 is /usr/bin/python3

# Check if variable is set
$ [ -z "$JAVA_HOME" ] && echo "JAVA_HOME not set"

# Debug PATH issues
$ echo $PATH | grep -o '[^:]*' | while read dir; do
    [ -d "$dir" ] || echo "Directory not found: $dir"
done
```

Environment Diagnostic Script

```
#!/bin/bash
# env_diagnostic.sh - Diagnose environment issues

echo "=== Environment Diagnostic ==="
```

```

echo "Date: $(date)"
echo "User: $USER"
echo "Shell: $SHELL"
echo "Bash Version: $BASH_VERSION"
echo

echo "=== PATH Analysis ==="
echo "PATH has $(echo $PATH | tr ':' '\n' | wc -l) directories"
echo $PATH | tr ':' '\n' | while read dir; do
    if [ -d "$dir" ]; then
        echo " $dir"
    else
        echo " $dir (not found)"
    fi
done
echo

echo "=== Important Variables ==="
for var in HOME USER SHELL LANG EDITOR PAGER; do
    echo "$var: ${!var:-'(not set)'}"
done

```

Understanding and properly managing the Bash environment is essential for creating robust scripts and maintaining a productive development environment. The environment affects everything from command execution to script behavior, making it a fundamental aspect of shell scripting.

Bash Features

Bash offers a rich set of features that make it powerful for both interactive use and scripting. Understanding these features will help you write more efficient and effective scripts.

Interactive Features

Command History

Bash maintains a history of commands you've executed:

```
# View command history
$ history
1001  ls -la
1002  cd /home
1003  pwd
1004  echo "hello world"
1005  history

# Execute previous command
$ !!

# Execute command by number
$ !1003

# Search history
$ !pw
pwd

# Reverse search (Ctrl+R)
(reverse-i-search)`ls': ls -la
```

History Configuration

```
# In ~/.bashrc
HISTSIZE=1000          # Commands in memory
HISTFILESIZE=2000      # Commands in history file
HISTCONTROL=ignoredups # Ignore duplicate commands
```

```
# Don't save certain commands
HISTIGNORE="ls:pwd:exit:clear"
```

Tab Completion

Bash provides intelligent tab completion:

```
# Complete commands
$ ec<TAB>
echo

# Complete file names
$ ls /usr/b<TAB>
bin/

# Complete variable names
$ echo $HO<TAB>
$HOME

# Complete options
$ ls --<TAB><TAB>
--all          --escape
--almost-all  --file-type
--author       --format
```

Custom Completion

```
# Add to ~/.bashrc
complete -W "start stop restart status" service
complete -d cd # Only directories for cd
```

Command Line Editing

Bash uses readline library for command editing:

```
# Cursor movement
Ctrl+A  # Beginning of line
Ctrl+E  # End of line
Ctrl+B  # Back one character
Ctrl+F  # Forward one character
Alt+B   # Back one word
Alt+F   # Forward one word
```

```
# Editing
Ctrl+D    # Delete character under cursor
Ctrl+H    # Delete character before cursor
Ctrl+K    # Kill from cursor to end of line
Ctrl+U    # Kill from cursor to beginning of line
Ctrl+W    # Kill word before cursor
Alt+D     # Kill word after cursor

# History navigation
Ctrl+P    # Previous command
Ctrl+N    # Next command
Ctrl+R    # Reverse search
```

Job Control

Bash provides sophisticated job control capabilities:

Background Jobs

```
# Run command in background
$ long_running_command &
[1] 12345

# List active jobs
$ jobs
[1]+  Running      long_running_command &

# Bring job to foreground
$ fg %1

# Send job to background
$ bg %1
```

Job Management

```
#!/bin/bash
# job_control_demo.sh

echo "Starting background jobs..."

# Start multiple background jobs
sleep 30 &
```

```

job1=$!

sleep 40 &
job2=$!

echo "Job 1 PID: $job1"
echo "Job 2 PID: $job2"

# Wait for specific job
wait $job1
echo "Job 1 completed"

# Wait for all background jobs
wait
echo "All jobs completed"

```

Process Control

```

# Suspend current process
Ctrl+Z

# Kill current process
Ctrl+C

# Send signals to processes
$ kill -TERM 12345
$ kill -KILL 12345
$ killall process_name

```

Aliases and Functions

Aliases

Simple command shortcuts:

```

# Define aliases
alias ll='ls -la'
alias la='ls -A'
alias l='ls -CF'
alias grep='grep --color=auto'

# Use aliases
$ ll
total 24

```



```

drwxr-xr-x  3 user user 4096 Jan 15 10:30 .
drwxr-xr-x 25 user user 4096 Jan 15 10:25 ..

# List all aliases
$ alias

# Remove alias
$ unalias ll

```

Functions

More complex reusable code:

```

# Simple function
greet() {
    echo "Hello, $1!"
}

# Function with multiple parameters
backup_file() {
    local file="$1"
    local backup_dir="$2"

    if [ -f "$file" ]; then
        cp "$file" "$backup_dir/${basename "$file"}.bak"
        echo "Backed up $file"
    else
        echo "File $file not found"
        return 1
    fi
}

# Use functions
$ greet "World"
Hello, World!

$ backup_file "/etc/hosts" "/backup"
Backed up /etc/hosts

```

Variable Features

Variable Types

```
# Regular variables
name="John"
age=25

# Arrays
fruits=("apple" "banana" "orange")
echo ${fruits[0]} # apple
echo ${fruits[@]} # all elements

# Associative arrays (Bash 4+)
declare -A colors
colors[red]="#FF0000"
colors[green]="#00FF00"
echo ${colors[red]} # #FF0000
```

Parameter Expansion

```
filename="document.txt"

# Basic expansion
echo $filename          # document.txt
echo ${filename}        # document.txt

# Length
echo ${#filename}       # 12

# Substring
echo ${filename:0:8}    # document

# Remove from end
echo ${filename%.*}     # document

# Remove from beginning
echo ${filename##*.}    # txt

# Replace
echo ${filename/txt/pdf} # document.pdf
```

Default Values

```
# Use default if variable is unset
echo ${name:-"Anonymous"}

# Set default if variable is unset
echo ${name:="Default Name"}

# Error if variable is unset
echo ${name:? "Variable name is required"}

# Use alternative if variable is set
echo ${name:+ "Name is set"}
```

Pattern Matching and Globbing

Basic Wildcards

```
# Match any characters
$ ls *.txt
file1.txt  file2.txt  document.txt

# Match single character
$ ls file?.txt
file1.txt  file2.txt

# Match character ranges
$ ls file[1-3].txt
file1.txt  file2.txt  file3.txt

# Match character sets
$ ls file[abc].txt
filea.txt  fileb.txt  filec.txt
```

Extended Globbing

```
# Enable extended globbing
shopt -s extglob

# Match one of several patterns
$ ls *.(txt|pdf|doc)

# Match zero or one occurrence
```

```
$ ls file?(s).txt

# Match zero or more occurrences
$ ls file*(s).txt

# Match one or more occurrences
$ ls file+(s).txt

# Match anything except pattern
$ ls !(*.tmp)
```

Globstar (Bash 4+)

```
# Enable globstar
shopt -s globstar

# Recursive matching
$ ls **/*.txt
dir1/file1.txt
dir1/subdir/file2.txt
dir2/file3.txt
```

Brace Expansion

Sequence Generation

```
# Number sequences
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10

$ echo {01..10}
01 02 03 04 05 06 07 08 09 10

# Letter sequences
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# Step sequences (Bash 4+)
$ echo {1..10..2}
1 3 5 7 9
```

String Expansion

```
# Multiple strings
$ echo {cat,dog,bird}
cat dog bird

# Combinations
$ echo {a,b}-{1,2}
a1 a2 b1 b2

# File operations
$ cp file.txt{,.bak} # Same as: cp file.txt file.txt.bak
$ mkdir -p project/{src,docs,tests}
```

Process Substitution

Input Substitution

```
# Compare output of two commands
$ diff <(ls dir1) <(ls dir2)

# Use command output as input
$ while read line; do
    echo "Processing: $line"
done <<(find /var/log -name "*.log")
```

Output Substitution

```
# Send output to multiple commands
$ echo "data" | tee >(command1) >(command2)

# Log to file and display
$ make 2>&1 | tee >(logger -t build)
```

Command Substitution

Modern Syntax

```
# Preferred method
current_date=$(date)
file_count=$(ls | wc -l)
user_home=$(eval echo ~$USER)

echo "Today is $current_date"
echo "Files in directory: $file_count"
```

Legacy Syntax

```
# Older method (still works)
current_date=`date`
file_count=`ls | wc -l`
```

Nested Substitution

```
# Complex nested commands
backup_name="backup_$(date +%Y%m%d)_$(hostname).tar.gz"
echo $backup_name # backup_20240115_myserver.tar.gz
```

Arithmetic Operations

Arithmetic Expansion

```
# Basic arithmetic
result=$((5 + 3))
echo $result # 8

# Variables in arithmetic
a=10
b=5
echo $((a + b)) # 15
echo $((a * b)) # 50
echo $((a / b)) # 2
echo $((a % b)) # 0
```

Arithmetic Commands

```
# let command
let "result = 5 + 3"
echo $result # 8

# expr command (external)
result=$(expr 5 + 3)
echo $result # 8
```

Advanced Arithmetic

```
# Increment/decrement
counter=0
((counter++))
echo $counter # 1

((counter += 5))
echo $counter # 6

# Comparison in arithmetic
if ((counter > 5)); then
    echo "Counter is greater than 5"
fi
```

Conditional Expressions

Test Command

```
# File tests
if [ -f "file.txt" ]; then
    echo "File exists"
fi

# String tests
if [ "$name" = "John" ]; then
    echo "Hello John"
fi

# Numeric tests
if [ $age -gt 18 ]; then
    echo "Adult"
fi
```

Extended Test

```
# Bash extended test
if [[ $name == "John" ]]; then
    echo "Hello John"
fi

# Pattern matching
if [[ $filename == *.txt ]]; then
    echo "Text file"
fi

# Regular expressions
if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi
```

I/O Redirection

Basic Redirection

```
# Redirect stdout
$ echo "Hello" > file.txt

# Redirect stderr
$ command 2> error.log

# Redirect both
$ command > output.log 2>&1
$ command &> output.log # Bash shortcut
```

Advanced Redirection

```
# Append to file
$ echo "More data" >> file.txt

# Here documents
$ cat << EOF > config.txt
Setting1=value1
Setting2=value2
EOF

# Here strings
```



```
$ grep "pattern" <<< "$variable"
```

Shell Options

Set Options

```
# Exit on error
set -e

# Exit on undefined variable
set -u

# Exit on pipe failure
set -o pipefail

# Combine options
set -euo pipefail
```

Shopt Options

```
# Enable extended globbing
shopt -s extglob

# Enable case-insensitive globbing
shopt -s nocaseglob

# Enable recursive globbing
shopt -s globstar

# List all options
shopt
```

These features make Bash a powerful and flexible shell for both interactive use and scripting. Understanding and utilizing these features will significantly improve your productivity and the quality of your scripts.

What is Bash?

Bash (Bourne Again Shell) is the most widely used shell in Linux and Unix systems. It's an enhanced version of the original Bourne shell (sh) with many additional features that make it powerful for both interactive use and scripting.

History and Background

The Evolution

1. **Bourne Shell (sh)** - 1977: Original Unix shell by Stephen Bourne
2. **C Shell (csh)** - 1978: Added command history and job control
3. **Korn Shell (ksh)** - 1983: Combined features of sh and csh
4. **Bash** - 1989: GNU's "Bourne Again Shell" by Brian Fox

Why "Bourne Again"?

- **Pun:** Play on "born again" (reborn)
- **Compatibility:** Maintains compatibility with original Bourne shell
- **Enhancement:** Adds modern features while preserving classic functionality

Bash Characteristics

POSIX Compliance

Bash is largely POSIX-compliant, meaning scripts written for POSIX shells should work in Bash:

```
#!/bin/sh
# This POSIX script works in bash
echo "Hello, World!"
if [ -f "/etc/passwd" ]; then
    echo "Password file exists"
fi
```

GNU Project

- **Free Software:** Part of the GNU operating system
- **Open Source:** Source code freely available
- **Community Driven:** Developed and maintained by volunteers

Bash Versions

Check Your Bash Version

```
$ bash --version
GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu)

# Or from within bash
$ echo $BASH_VERSION
5.1.16(1)-release

# Short version
$ echo ${BASH_VERSION%%.*}
5
```

Major Version Differences

Bash 3.x (2004-2009)

- Regular expressions in conditional expressions
- Process substitution improvements

Bash 4.x (2009-2020)

- Associative arrays
- Case modification operators
- Globstar (**) pattern matching

Bash 5.x (2019-present)

- Improved performance
- New variable expansion features
- Enhanced debugging capabilities

Version-Specific Features

```
#!/bin/bash

# Check bash version for feature compatibility
if [ "${BASH_VERSION%%.*}" -ge 4 ]; then
    # Bash 4+ features
    declare -A assoc_array
    assoc_array[key]="value"
    echo "Using associative arrays"
else
    # Fallback for older bash
    echo "Using indexed arrays"
fi
```

Bash vs Other Shells

Bash vs Bourne Shell (sh)

```
# Bash-specific features not in sh
echo ${BASH_VERSION}      # Bash version variable
echo ${!var*}             # Variable name expansion
declare -A array          # Associative arrays
[[ $var =~ regex ]]      # Regular expression matching
```

Bash vs Zsh

```
# Bash
$ echo $SHELL
/bin/bash

# Zsh has more advanced features
$ echo $SHELL
/usr/bin/zsh
```

Bash vs Fish

```
# Bash syntax
if [ $status -eq 0 ]; then
    echo "Success"
fi
```

```
# Fish syntax (different)
if test $status -eq 0
    echo "Success"
end
```

Where Bash is Used

Default Shell

Bash is the default shell on most Linux distributions:

```
# Check default shell
$ echo $SHELL
/bin/bash

# Check available shells
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
```

System Scripts

Many system scripts use Bash:

```
# System initialization scripts
$ head -1 /etc/init.d/ssh
#!/bin/bash

# Package management scripts
$ head -1 /usr/bin/apt-get
#!/bin/bash
```

Development and DevOps

```
# Build scripts
#!/bin/bash
make clean
make all
```

```
make install

# Deployment scripts
#!/bin/bash
git pull origin main
npm install
npm run build
systemctl restart myapp
```

Bash Capabilities

Interactive Features

```
# Command history
$ history | tail -5
1001  ls -la
1002  cd /home
1003  pwd
1004  echo "hello"
1005  history | tail -5

# Tab completion
$ ls /usr/b<TAB>
bin/

# Command editing
# Use arrow keys, Ctrl+A (beginning), Ctrl+E (end)
```

Programming Features

```
#!/bin/bash

# Variables
name="John"
age=25

# Arrays
fruits=("apple" "banana" "orange")

# Functions
greet() {
    echo "Hello, $1!"
}
```

```

# Control structures
if [ $age -gt 18 ]; then
    echo "Adult"
fi

# Loops
for fruit in "${fruits[@]}"; do
    echo "Fruit: $fruit"
done

```

Advanced Features

```

#!/bin/bash

# Process substitution
diff <(ls dir1) <(ls dir2)

# Parameter expansion
filename="document.txt"
echo ${filename%.*}      # document
echo ${filename##*.}     # txt

# Regular expressions
if [[ $email =~ ^[a-zA-Z0-9._%+~]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi

```

Bash Configuration

Configuration Files

```

# System-wide configuration
/etc/bash.bashrc      # All users
/etc/profile           # Login shells

# User-specific configuration
~/.bashrc              # Interactive non-login shells
~/.bash_profile        # Login shells
~/.bash_logout         # Logout cleanup

```

Customization Example

```
# ~/.bashrc
# Custom prompt
PS1='\u@\h:\w\$ '

# Aliases
alias ll='ls -la'
alias grep='grep --color=auto'

# Functions
mkcd() {
    mkdir -p "$1" && cd "$1"
}

# Environment variables
export EDITOR=vim
export PATH="$HOME/bin:$PATH"
```

Bash Built-in Commands

Bash includes many built-in commands for efficiency:

```
# File operations
$ type cd
cd is a shell builtin

$ type echo
echo is a shell builtin

# List all built-ins
$ help
# or
$ compgen -b
```

Common Built-ins

```
# Navigation
cd /path/to/directory
pwd

# Variables
export VAR=value
unset VAR
```



```
# Control
exit 0
return 1

# Information
type command
which command
help command
```

Bash Scripting Advantages

1. Ubiquity

- Available on virtually all Unix-like systems
- Default shell on most Linux distributions
- Consistent behavior across platforms

2. Integration

- Seamless integration with system commands
- Easy to call external programs
- Natural for system administration tasks

3. Learning Curve

- Familiar to anyone who uses the command line
- Gradual learning from simple commands to complex scripts
- Extensive documentation and community support

4. Power and Flexibility

```
#!/bin/bash
# Powerful one-liners
find /var/log -name "*.log" -mtime +7 -exec rm {} \;

# Complex data processing
awk '/ERROR/ {count++} END {print "Errors:", count}' logfile.txt

# System monitoring
ps aux | awk '$3 > 80 {print $2, $11}' | head -10
```

When to Use Bash

Perfect For:

- **System administration:** User management, backups, monitoring
- **DevOps tasks:** Deployment, CI/CD pipelines
- **File processing:** Batch operations, log analysis
- **Glue scripts:** Connecting different tools and programs
- **Quick automation:** Tasks that don't require complex logic

Consider Alternatives For:

- **Complex applications:** Use Python, Go, or other languages
- **Performance-critical tasks:** Compiled languages are faster
- **Cross-platform GUI apps:** Use appropriate frameworks
- **Heavy mathematical computations:** Use specialized tools

Getting Started with Bash

Check if Bash is Available

```
$ which bash
/bin/bash

$ bash --version
GNU bash, version 5.1.16(1)-release
```

Start Using Bash

```
# If not your default shell
$ bash

# Make bash your default shell
$ chsh -s $(which bash)
```

First Bash Script

```
#!/bin/bash
echo "Welcome to Bash scripting!"
echo "Bash version: $BASH_VERSION"
echo "Today is: $(date)"
```

Bash is more than just a command interpreter—it's a powerful programming environment that bridges the gap between simple command-line operations and full-featured programming languages. Its widespread adoption, extensive features, and excellent documentation make it an essential tool for anyone working with Linux or Unix systems.

Variables

Variable Basics

Variables are fundamental building blocks in Bash scripting. They allow you to store data, make scripts flexible, and create reusable code. Understanding how to work with variables effectively is essential for writing robust shell scripts.

What are Variables?

Variables are named storage locations that hold data. In Bash, variables can store: - Text strings - Numbers (treated as strings) - Command output - File paths - Configuration values

Variable Declaration and Assignment

Basic Syntax

```
# Variable assignment (no spaces around =)
variable_name="value"

# Examples
name="John Doe"
age=25
city="New York"
is_active=true
```

Important Rules

1. **No spaces** around the equals sign
2. **Case sensitive:** name and Name are different variables
3. **No type declaration** needed
4. **Quote values** with spaces or special characters

Common Mistakes

```
# Wrong - spaces around =
name = "John"          # Error!

# Wrong - unquoted value with spaces
name=John Doe          # Error!

# Correct
name="John Doe"        # Correct
```

Variable Naming Conventions

Valid Variable Names

```
# Good variable names
username="john"
user_name="john"
userName="john"
USER_NAME="john"
file1="document.txt"
_private_var="secret"
```

Invalid Variable Names

```
# Invalid - cannot start with number
1name="john"          # Error!

# Invalid - contains special characters
user-name="john"      # Error!
user@name="john"      # Error!

# Invalid - reserved words
for="value"           # Error!
if="value"            # Error!
```

Best Practices

```
# Use descriptive names
config_file="/etc/myapp.conf"
max_connections=100
database_host="localhost"
```

```
# Use uppercase for constants
readonly PI=3.14159
readonly MAX_RETRIES=3

# Use lowercase for regular variables
current_user="john"
temp_directory="/tmp"
```

Accessing Variable Values

Basic Access

```
name="Alice"

# Access variable value
echo $name           # Alice
echo ${name}         # Alice (preferred for clarity)
```

When to Use Braces

```
name="Alice"

# Without braces - can be ambiguous
echo $names          # Empty (variable 'names' doesn't exist)

# With braces - clear separation
echo ${name}s        # Alices

# Concatenation examples
prefix="file"
echo ${prefix}_backup.txt # file_backup.txt
echo $prefix_backup.txt  # Empty (looks for variable 'prefix_backup')
```

Variable Types and Behavior

String Variables

```
# Simple strings
greeting="Hello"
message="Welcome to Bash scripting"

# Strings with special characters
path="/home/user/My Documents"
command="ls -la"

# Empty strings
empty_var=""
another_empty=""
```

Numeric Variables

```
# Numbers are stored as strings but can be used in arithmetic
count=10
price=19.99
year=2024

# Arithmetic operations
total=$((count * 2))
echo $total          # 20

# Comparison
if [ $count -gt 5 ]; then
    echo "Count is greater than 5"
fi
```

Boolean-like Variables

```
# Bash doesn't have true boolean types
# Use strings to represent boolean values
is_enabled="true"
debug_mode="false"
has_permission="yes"

# Check boolean-like values
if [ "$is_enabled" = "true" ]; then
    echo "Feature is enabled"
fi
```


Variable Scope

Global Variables

```
#!/bin/bash
# Global variable - accessible everywhere
global_var="I'm global"

function test_function() {
    echo "Inside function: $global_var"
}

echo "Outside function: $global_var"
test_function
```

Local Variables

```
#!/bin/bash
function test_local() {
    local local_var="I'm local"
    global_var="Modified global"

    echo "Local variable: $local_var"
    echo "Global variable: $global_var"
}

global_var="Original global"
echo "Before function: $global_var"

test_local

echo "After function: $global_var"
echo "Local variable outside: $local_var" # Empty
```

Reading User Input

Basic Input

```
#!/bin/bash
echo "What's your name?"
read name
echo "Hello, $name!"
```

Input with Prompt

```
#!/bin/bash
read -p "Enter your age: " age
echo "You are $age years old"
```

Multiple Variables

```
#!/bin/bash
echo "Enter first and last name:"
read first_name last_name
echo "Hello, $first_name $last_name!"
```

Reading into Array

```
#!/bin/bash
echo "Enter three colors:"
read -a colors
echo "First color: ${colors[0]}"
echo "All colors: ${colors[@]}"
```

Silent Input (Passwords)

```
#!/bin/bash
read -s -p "Enter password: " password
echo # New line after hidden input
echo "Password length: ${#password}"
```

Variable Assignment Methods

Direct Assignment

```
name="John"
age=25
```

Command Substitution

```
# Modern syntax (preferred)
current_date=$(date)
file_count=$(ls | wc -l)
current_user=$(whoami)

# Legacy syntax (still works)
current_date=`date`
file_count=`ls | wc -l`
```

Reading from Files

```
# Read first line of file
first_line=$(head -n1 /etc/passwd)

# Read entire file content
file_content=$(cat /etc/hostname)

# Read with error handling
if config=$(cat config.txt 2>/dev/null); then
    echo "Config loaded: $config"
else
    echo "Failed to read config file"
fi
```

Arithmetic Assignment

```
# Simple arithmetic
count=5
count=$((count + 1))
echo $count          # 6

# Using let command
let count=count+1
echo $count          # 7

# Increment/decrement
((count++))
echo $count          # 8
```

Variable Manipulation

String Length

```
text="Hello World"
echo ${#text}      # 11
```

Substring Extraction

```
text="Hello World"

# Extract substring: ${variable:start:length}
echo ${text:0:5}    # Hello
echo ${text:6}      # World
echo ${text:6:3}    # Wor
echo ${text:-5}     # World (note the space before -)
```

Case Conversion

```
text="Hello World"

# Convert to uppercase (Bash 4+)
echo ${text^^}      # HELLO WORLD

# Convert to lowercase (Bash 4+)
echo ${text,,}      # hello world

# Convert first character
echo ${text^}       # Hello World
echo ${text,}       # hello World
```

Pattern Replacement

```
filename="document.txt"

# Replace first occurrence
echo ${filename/txt/pdf}    # document.pdf

# Replace all occurrences
echo ${filename//t/T}      # documentT.TxT

# Remove from beginning
```

```
echo ${filename#doc}          # ument.txt

# Remove from end
echo ${filename%.txt}         # document
```

Default Values and Error Handling

Default Values

```
# Use default if variable is unset or empty
name=${name:-"Anonymous"}
echo "Hello, $name"

# Set default if variable is unset
name=${name:="Default Name"}

# Use alternative if variable is set
message=${name:+ "Hello, $name"}
```

Error on Unset Variables

```
# Exit with error if variable is unset
required_var=${required_var:? "Variable required_var must be set"}

# Custom error message
database_host=${database_host:? "Error: DATABASE_HOST environment variable is required"}
```

Practical Example

```
#!/bin/bash
# Configuration script with defaults

# Set defaults
config_file=${CONFIG_FILE:-"/etc/myapp.conf"}
log_level=${LOG_LEVEL:-"INFO"}
max_connections=${MAX_CONNECTIONS:-"100"}

echo "Configuration:"
echo "Config file: $config_file"
echo "Log level: $log_level"
echo "Max connections: $max_connections"
```

```
# Validate required variables
database_url=${DATABASE_URL:? "Error: DATABASE_URL must be set"}
api_key=${API_KEY:? "Error: API_KEY must be set"}
```

Variable Arrays (Preview)

Simple Array

```
# Create array
fruits=("apple" "banana" "orange")

# Access elements
echo ${fruits[0]}    # apple
echo ${fruits[1]}    # banana

# All elements
echo ${fruits[@]}    # apple banana orange

# Array length
echo ${#fruits[@]}    # 3
```

Best Practices

1. Always Quote Variables

```
# Good - prevents word splitting
if [ "$name" = "John Doe" ]; then
    echo "Hello, $name"
fi

# Bad - can break with spaces
if [ $name = "John Doe" ]; then
    echo "Hello, $name"
fi
```

2. Use Meaningful Names

```
# Good
database_connection_string="mysql://localhost:3306/mydb"
max_retry_attempts=3
```

```
# Bad
dcs="mysql://localhost:3306/mydb"
mra=3
```

3. Initialize Variables

```
# Good - initialize variables
count=0
total=0
error_message=""

# Process data
for file in *.txt; do
    ((count++))
    # ... processing ...
done
```

4. Use readonly for Constants

```
# Constants should be readonly
readonly PI=3.14159
readonly MAX_USERS=1000
readonly CONFIG_DIR="/etc/myapp"

# This will cause an error
# PI=3.14 # Error: PI is readonly
```

5. Validate Input

```
#!/bin/bash
read -p "Enter a number: " number

# Validate input
if [[ ! $number =~ ^[0-9]+$ ]]; then
    echo "Error: Please enter a valid number"
    exit 1
fi

echo "You entered: $number"
```

Common Variable Patterns

Configuration Variables

```
#!/bin/bash
# Application configuration

# Default configuration
APP_NAME="MyApplication"
APP_VERSION="1.0.0"
CONFIG_FILE="/etc/${APP_NAME,,}.conf"
LOG_FILE="/var/log/${APP_NAME,,}.log"
PID_FILE="/var/run/${APP_NAME,,}.pid"

# Load configuration file if it exists
if [ -f "$CONFIG_FILE" ]; then
    source "$CONFIG_FILE"
fi
```

Temporary Variables

```
#!/bin/bash
# Temporary file handling

temp_dir=$(mktemp -d)
temp_file=$(mktemp)

# Cleanup function
cleanup() {
    rm -rf "$temp_dir"
    rm -f "$temp_file"
}

# Set trap to cleanup on exit
trap cleanup EXIT

# Use temporary files
echo "Processing data..." > "$temp_file"
```

Understanding variable basics is crucial for Bash scripting. Variables make your scripts flexible, maintainable, and powerful. Practice these concepts and patterns to build a solid foundation for more advanced scripting techniques.

Control

Conditional Statements

Conditional statements allow your scripts to make decisions and execute different code paths based on conditions. They are essential for creating intelligent, responsive scripts that can handle various scenarios.

The if Statement

The `if` statement is the most basic conditional construct in Bash.

Basic Syntax

```
if [ condition ]; then
    # commands to execute if condition is true
fi
```

Simple Example

```
#!/bin/bash
age=20

if [ $age -ge 18 ]; then
    echo "You are an adult"
fi
```

if-else Statement

Add an alternative path when the condition is false:

```
#!/bin/bash
age=16

if [ $age -ge 18 ]; then
    echo "You are an adult"
else
    echo "You are a minor"
```

```
fi
```

if-elif-else Statement

Handle multiple conditions:

```
#!/bin/bash
score=85

if [ $score -ge 90 ]; then
    echo "Grade: A"
elif [ $score -ge 80 ]; then
    echo "Grade: B"
elif [ $score -ge 70 ]; then
    echo "Grade: C"
elif [ $score -ge 60 ]; then
    echo "Grade: D"
else
    echo "Grade: F"
fi
```

Test Conditions

Numeric Comparisons

```
#!/bin/bash
num1=10
num2=20

# Equal
if [ $num1 -eq $num2 ]; then
    echo "Numbers are equal"
fi

# Not equal
if [ $num1 -ne $num2 ]; then
    echo "Numbers are not equal"
fi

# Greater than
if [ $num1 -gt $num2 ]; then
    echo "$num1 is greater than $num2"
fi
```

```

# Less than
if [ $num1 -lt $num2 ]; then
    echo "$num1 is less than $num2"
fi

# Greater than or equal
if [ $num1 -ge $num2 ]; then
    echo "$num1 is greater than or equal to $num2"
fi

# Less than or equal
if [ $num1 -le $num2 ]; then
    echo "$num1 is less than or equal to $num2"
fi

```

String Comparisons

```

#!/bin/bash
name1="Alice"
name2="Bob"
empty_string=""

# String equality
if [ "$name1" = "$name2" ]; then
    echo "Names are the same"
else
    echo "Names are different"
fi

# String inequality
if [ "$name1" != "$name2" ]; then
    echo "Names are different"
fi

# String length (non-zero)
if [ -n "$name1" ]; then
    echo "Name1 is not empty"
fi

# String length (zero)
if [ -z "$empty_string" ]; then
    echo "String is empty"
fi

# Lexicographic comparison

```

```
if [[ "$name1" < "$name2" ]]; then
    echo "$name1 comes before $name2 alphabetically"
fi
```

File Tests

```
#!/bin/bash
filename="test.txt"
directory="mydir"

# File exists
if [ -e "$filename" ]; then
    echo "File exists"
fi

# Regular file
if [ -f "$filename" ]; then
    echo "It's a regular file"
fi

# Directory
if [ -d "$directory" ]; then
    echo "It's a directory"
fi

# Readable
if [ -r "$filename" ]; then
    echo "File is readable"
fi

# Writable
if [ -w "$filename" ]; then
    echo "File is writable"
fi

# Executable
if [ -x "$filename" ]; then
    echo "File is executable"
fi

# File size greater than zero
if [ -s "$filename" ]; then
    echo "File is not empty"
fi
```

```
# Symbolic link
if [ -L "$filename" ]; then
    echo "It's a symbolic link"
fi
```

Advanced Test Constructs

Double Brackets [[]]

The [[]] construct provides more features than single brackets:

```
#!/bin/bash
name="Alice"
number="123"

# Pattern matching
if [[ $name == A* ]]; then
    echo "Name starts with A"
fi

# Regular expressions
if [[ $number =~ ^[0-9]+$ ]]; then
    echo "It's a number"
fi

# Multiple conditions with && and ||
if [[ $name == "Alice" && $number -gt 100 ]]; then
    echo "Name is Alice and number is greater than 100"
fi

if [[ $name == "Alice" || $name == "Bob" ]]; then
    echo "Name is either Alice or Bob"
fi
```

Arithmetic Evaluation (())

For numeric comparisons, you can use arithmetic evaluation:

```
#!/bin/bash
x=10
y=20

if (( x < y )); then
    echo "$x is less than $y"
```

```

fi

if (( x + y == 30 )); then
    echo "Sum is 30"
fi

if (( x % 2 == 0 )); then
    echo "$x is even"
fi

```

Logical Operators

AND Operator (&&)

```

#!/bin/bash
age=25
has_license=true

if [ $age -ge 18 ] && [ "$has_license" = "true" ]; then
    echo "Can drive"
fi

# Alternative syntax
if [[ $age -ge 18 && $has_license == "true" ]]; then
    echo "Can drive"
fi

```

OR Operator (||)

```

#!/bin/bash
day="Saturday"

if [ "$day" = "Saturday" ] || [ "$day" = "Sunday" ]; then
    echo "It's weekend!"
fi

# Alternative syntax
if [[ $day == "Saturday" || $day == "Sunday" ]]; then
    echo "It's weekend!"
fi

```

NOT Operator (!)

```
#!/bin/bash
filename="test.txt"

# Negate condition
if [ ! -f "$filename" ]; then
    echo "File does not exist"
fi

# Alternative
if ! [ -f "$filename" ]; then
    echo "File does not exist"
fi
```

Practical Examples

File Backup Script

```
#!/bin/bash
source_file="$1"
backup_dir="/backup"

# Check if source file is provided
if [ $# -eq 0 ]; then
    echo "Usage: $0 <source_file>"
    exit 1
fi

# Check if source file exists
if [ ! -f "$source_file" ]; then
    echo "Error: Source file '$source_file' does not exist"
    exit 1
fi

# Check if backup directory exists
if [ ! -d "$backup_dir" ]; then
    echo "Creating backup directory..."
    mkdir -p "$backup_dir"
fi

# Create backup
backup_name="$(basename "$source_file").backup.$(date +%Y%m%d_%H%M%S)"
if cp "$source_file" "$backup_dir/$backup_name"; then
```



```

        echo "Backup created: $backup_dir/$backup_name"
    else
        echo "Error: Failed to create backup"
        exit 1
    fi
fi

```

System Health Check

```

#!/bin/bash
# System health check script

echo "=== System Health Check ==="

# Check disk usage
disk_usage=$(df / | awk 'NR==2 {print $5}' | sed 's/%//')
if [ $disk_usage -gt 90 ]; then
    echo "WARNING: Disk usage is ${disk_usage}%"
elif [ $disk_usage -gt 80 ]; then
    echo "CAUTION: Disk usage is ${disk_usage}%"
else
    echo "OK: Disk usage is ${disk_usage}%"
fi

# Check memory usage
memory_usage=$(free | awk 'NR==2{printf "%.0f", $3*100/$2}')
if [ $memory_usage -gt 90 ]; then
    echo "WARNING: Memory usage is ${memory_usage}%"
else
    echo "OK: Memory usage is ${memory_usage}%"
fi

# Check if critical services are running
services=("ssh" "nginx" "mysql")
for service in "${services[@]}; do
    if systemctl is-active --quiet "$service"; then
        echo "OK: $service is running"
    else
        echo "ERROR: $service is not running"
    fi
done

```

User Input Validation

```
#!/bin/bash
# User registration script with validation

echo "User Registration"
echo "===== "

# Get username
while true; do
    read -p "Enter username (3-20 characters, alphanumeric only): " username

    if [[ -z "$username" ]]; then
        echo "Username cannot be empty"
    elif [[ ${#username} -lt 3 ]]; then
        echo "Username must be at least 3 characters"
    elif [[ ${#username} -gt 20 ]]; then
        echo "Username must be no more than 20 characters"
    elif [[ ! $username =~ ^[a-zA-Z0-9]+$ ]]; then
        echo "Username can only contain letters and numbers"
    else
        break
    fi
done

# Get email
while true; do
    read -p "Enter email address: " email

    if [[ -z "$email" ]]; then
        echo "Email cannot be empty"
    elif [[ ! $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
        echo "Please enter a valid email address"
    else
        break
    fi
done

# Get age
while true; do
    read -p "Enter age (18-120): " age

    if [[ ! $age =~ ^[0-9]+$ ]]; then
        echo "Age must be a number"
    elif [ $age -lt 18 ]; then
        echo "You must be at least 18 years old"
```

```

        elif [ $age -gt 120 ]; then
            echo "Please enter a realistic age"
        else
            break
        fi
    done

    echo
    echo "Registration successful!"
    echo "Username: $username"
    echo "Email: $email"
    echo "Age: $age"

```

Nested Conditionals

You can nest if statements for complex logic:

```

#!/bin/bash
weather="sunny"
temperature=75

if [ "$weather" = "sunny" ]; then
    if [ $temperature -gt 70 ]; then
        echo "Perfect day for outdoor activities!"
    else
        echo "Sunny but a bit cold"
    fi
elif [ "$weather" = "rainy" ]; then
    if [ $temperature -gt 60 ]; then
        echo "Warm rain, good for plants"
    else
        echo "Cold and rainy, stay inside"
    fi
else
    echo "Weather is $weather"
fi

```

Short-Circuit Evaluation

Use `&&` and `||` for concise conditional execution:

```

#!/bin/bash
filename="test.txt"

```

```
# Execute command only if condition is true
[ -f "$filename" ] && echo "File exists"

# Execute command only if condition is false
[ ! -f "$filename" ] || echo "File exists"

# Chain multiple conditions
[ -f "$filename" ] && [ -r "$filename" ] && echo "File exists and is readable"

# Alternative to if-else
[ $# -eq 0 ] && echo "No arguments provided" || echo "Arguments provided"
```

Common Pitfalls and Best Practices

1. Always Quote Variables

```
# Wrong - can break with spaces
if [ $name = "John Doe" ]; then
    echo "Hello John"
fi

# Correct
if [ "$name" = "John Doe" ]; then
    echo "Hello John"
fi
```

2. Use [[]] for String Operations

```
# Better for pattern matching and regex
if [[ $filename == *.txt ]]; then
    echo "Text file"
fi

if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi
```

3. Handle Empty Variables

```
# Safe way to handle potentially empty variables
if [ -n "${name:-}" ]; then
    echo "Name is set: $name"
else
    echo "Name is not set"
fi
```

4. Use Meaningful Conditions

```
# Good - readable and clear
if [ $user_age -ge $minimum_age ]; then
    grant_access
fi

# Bad - unclear magic numbers
if [ $age -ge 21 ]; then
    do_something
fi
```

5. Exit Codes for Error Handling

```
#!/bin/bash
if ! command_that_might_fail; then
    echo "Command failed"
    exit 1
fi

# Or check exit code explicitly
command_that_might_fail
if [ $? -ne 0 ]; then
    echo "Command failed with exit code $?"
    exit 1
fi
```

Conditional statements are the foundation of decision-making in shell scripts. Master these concepts to create scripts that can intelligently respond to different situations and handle various scenarios gracefully.

Loops

Loops allow you to execute a block of code repeatedly, making them essential for automating repetitive tasks. Bash provides several types of loops, each suited for different scenarios.

for Loops

The `for` loop is the most commonly used loop in Bash scripting.

Basic for Loop Syntax

```
for variable in list; do
    # commands
done
```

Simple Examples

```
#!/bin/bash

# Loop through a list of items
for fruit in apple banana orange; do
    echo "I like $fruit"
done

# Loop through numbers
for number in 1 2 3 4 5; do
    echo "Number: $number"
done
```

Looping Through Files

```
#!/bin/bash

# Process all .txt files in current directory
for file in *.txt; do
    if [ -f "$file" ]; then
```

```

        echo "Processing: $file"
        wc -l "$file"
    fi
done

# Process files in a specific directory
for file in /var/log/*.log; do
    if [ -f "$file" ]; then
        echo "Log file: $(basename "$file")"
        echo "Size: $(du -h "$file" | cut -f1)"
    fi
done

```

C-style for Loop

```

#!/bin/bash

# C-style for loop
for ((i=1; i<=10; i++)); do
    echo "Iteration: $i"
done

# Counting backwards
for ((i=10; i>=1; i--)); do
    echo "Countdown: $i"
done

# Step by 2
for ((i=0; i<=20; i+=2)); do
    echo "Even number: $i"
done

```

Using Brace Expansion

```

#!/bin/bash

# Loop through a range of numbers
for i in {1..10}; do
    echo "Number: $i"
done

# Loop with step
for i in {1..20..2}; do
    echo "Odd number: $i"
done

```

```
done

# Loop through letters
for letter in {a..z}; do
    echo "Letter: $letter"
done
```

Looping Through Arrays

```
#!/bin/bash

# Define an array
colors=("red" "green" "blue" "yellow")

# Loop through array elements
for color in "${colors[@]}"; do
    echo "Color: $color"
done

# Loop through array indices
for i in "${!colors[@]}"; do
    echo "Index $i: ${colors[i]}"
done
```

while Loops

The `while` loop continues executing as long as a condition is true.

Basic while Loop

```
#!/bin/bash

counter=1
while [ $counter -le 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```


Reading File Line by Line

```
#!/bin/bash

filename="data.txt"

# Create sample file
cat > "$filename" << EOF
Line 1
Line 2
Line 3
EOF

# Read file line by line
while IFS= read -r line; do
    echo "Processing: $line"
done < "$filename"
```

Menu System with while Loop

```
#!/bin/bash

show_menu() {
    echo "=== Menu ==="
    echo "1. Show date"
    echo "2. Show users"
    echo "3. Show disk usage"
    echo "4. Exit"
    echo -n "Choose option: "
}

while true; do
    show_menu
    read choice

    case $choice in
        1)
            echo "Current date: $(date)"
            ;;
        2)
            echo "Logged in users:"
            who
            ;;
        3)
            echo "Disk usage:"
    esac
done
```

```

        df -h
        ;;
    4)
        echo "Goodbye!"
        break
        ;;
    *)
        echo "Invalid option. Please try again."
        ;;
esac

echo
read -p "Press Enter to continue..."
echo
done

```

Monitoring with while Loop

```

#!/bin/bash

# Monitor system load
echo "Monitoring system load (Ctrl+C to stop)..."

while true; do
    load=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}' | sed 's/,//')
    echo "$(date): Load average: $load"

    # Alert if load is high
    if (( $(echo "$load > 2.0" | bc -l) )); then
        echo "WARNING: High system load!"
    fi

    sleep 5
done

```

until Loops

The until loop continues executing until a condition becomes true (opposite of while).

Basic until Loop

```
#!/bin/bash

counter=1
until [ $counter -gt 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```

Waiting for a Service

```
#!/bin/bash

service_name="nginx"

echo "Waiting for $service_name to start..."
until systemctl is-active --quiet "$service_name"; do
    echo "Service not ready, waiting..."
    sleep 2
done

echo "$service_name is now running!"
```

Waiting for File

```
#!/bin/bash

target_file="/tmp/ready.flag"

echo "Waiting for file $target_file to appear..."
until [ -f "$target_file" ]; do
    echo "File not found, waiting..."
    sleep 1
done

echo "File found! Continuing..."
```

Loop Control Statements

break Statement

Exits the loop immediately:

```
#!/bin/bash

# Find first .txt file
for file in *; do
    if [[ "$file" == *.txt ]]; then
        echo "Found text file: $file"
        break
    fi
    echo "Checking: $file"
done
```

continue Statement

Skips the rest of the current iteration:

```
#!/bin/bash

# Process only .txt files
for file in *; do
    # Skip non-txt files
    if [[ "$file" != *.txt ]]; then
        continue
    fi

    echo "Processing text file: $file"
    wc -l "$file"
done
```

Nested Loops with break and continue

```
#!/bin/bash

# Nested loops example
for i in {1..3}; do
    echo "Outer loop: $i"

    for j in {1..5}; do
        if [ $j -eq 3 ]; then
```

```

        echo "  Skipping inner loop iteration $j"
        continue
    fi

    if [ $i -eq 2 ] && [ $j -eq 4 ]; then
        echo "  Breaking inner loop at $i,$j"
        break
    fi

    echo "  Inner loop: $j"
done
done

```

Practical Loop Examples

Batch File Processing

```

#!/bin/bash

# Batch resize images
image_dir="./images"
output_dir="./thumbnails"

# Create output directory
mkdir -p "$output_dir"

for image in "$image_dir"/*.{jpg,jpeg,png}; do
    # Skip if no files match
    [ ! -f "$image" ] && continue

    filename=$(basename "$image")
    name="${filename%.*}"
    extension="${filename##*.*}"

    echo "Processing: $filename"

    # Resize image (requires ImageMagick)
    if command -v convert >/dev/null 2>&1; then
        convert "$image" -resize 200x200 "$output_dir/${name}_thumb.$extension"
        echo "Created thumbnail: ${name}_thumb.$extension"
    else
        echo "ImageMagick not installed, skipping resize"
    fi
done

```

Log Analysis

```
#!/bin/bash

log_file="/var/log/apache2/access.log"
declare -A ip_count

# Count IP addresses in log file
while IFS= read -r line; do
    # Extract IP address (first field)
    ip=$(echo "$line" | awk '{print $1}')

    # Skip invalid IPs
    if [[ $ip =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
        ((ip_count[$ip]++))
    fi
done < "$log_file"

# Display top 10 IPs
echo "Top 10 IP addresses:"
for ip in "${!ip_count[@]}"; do
    echo "$ip: ${ip_count[$ip]}"
done | sort -k2 -nr | head -10
```

System Backup Script

```
#!/bin/bash

# Directories to backup
backup_dirs="/home/user/documents" "/home/user/projects" "/etc"
backup_destination="/backup/$(date +%Y%m%d)"
log_file="/var/log/backup.log"

# Create backup destination
mkdir -p "$backup_destination"

# Function to log messages
log_message() {
    echo "$(date +%Y-%m-%d %H:%M:%S) - $1" | tee -a "$log_file"
}

log_message "Starting backup process"

# Backup each directory
for dir in "${backup_dirs[@]}"; do
```

```

if [ -d "$dir" ]; then
    dir_name=$(basename "$dir")
    log_message "Backing up $dir"

    if tar -czf "$backup_destination/${dir_name}.tar.gz" "$dir"; then
        log_message "Successfully backed up $dir"
    else
        log_message "ERROR: Failed to backup $dir"
    fi
else
    log_message "WARNING: Directory $dir does not exist"
fi
done

log_message "Backup process completed"

# Clean up old backups (keep only last 7 days)
find /backup -name "20*" -type d -mtime +7 -exec rm -rf {} \;
log_message "Cleaned up old backups"

```

User Management Script

```

#!/bin/bash

# Create multiple users from a list
users_file="users.txt"

# Create sample users file
cat > "$users_file" << EOF
john:John Doe:developer
jane:Jane Smith:admin
bob:Bob Johnson:user
EOF

# Process each user
while IFS=':' read -r username fullname role; do
    # Skip empty lines and comments
    [[ -z "$username" || "$username" =~ ^# ]] && continue

    echo "Processing user: $username"

    # Check if user already exists
    if id "$username" >/dev/null 2>&1; then
        echo "User $username already exists, skipping"
        continue
    fi
done < "$users_file"

```

```

fi

# Create user
if useradd -m -c "$fullname" "$username"; then
    echo "Created user: $username ($fullname)"

    # Set initial password
    echo "$username:temp123" | chpasswd
    echo "Set temporary password for $username"

    # Add to appropriate group based on role
    case "$role" in
        "admin")
            usermod -aG sudo "$username"
            echo "Added $username to sudo group"
            ;;
        "developer")
            usermod -aG developers "$username" 2>/dev/null || echo "developers group not"
            ;;
    esac
else
    echo "Failed to create user: $username"
fi

done < "$users_file"

```

Loop Performance Tips

Avoid Unnecessary Command Substitution

```

# Slow - calls external command in each iteration
for i in {1..1000}; do
    date_str=$(date)
    echo "$i: $date_str"
done

# Fast - call command once
date_str=$(date)
for i in {1..1000}; do
    echo "$i: $date_str"
done

```


Use Built-in Commands When Possible

```
# Slow - external command
for file in *.txt; do
    lines=$(wc -l < "$file")
    echo "$file: $lines lines"
done

# Faster - but still external command per file
for file in *.txt; do
    while IFS= read -r line; do
        ((count++))
    done < "$file"
    echo "$file: $count lines"
    count=0
done
```

Process Files Efficiently

```
# Process multiple files with one command
find . -name "*.log" -print0 | while IFS= read -r -d '' file; do
    echo "Processing: $file"
    # Process file
done
```

Common Loop Patterns

Retry Pattern

```
retry_command() {
    local max_attempts=3
    local attempt=1

    while [ $attempt -le $max_attempts ]; do
        echo "Attempt $attempt of $max_attempts"

        if command_that_might_fail; then
            echo "Command succeeded"
            return 0
        fi

        echo "Command failed, retrying..."
        ((attempt++))
    done
}
```

```

        sleep 2
    done

    echo "Command failed after $max_attempts attempts"
    return 1
}

```

Progress Indicator

```

#!/bin/bash

total_items=100

for ((i=1; i<=total_items; i++)); do
    # Simulate work
    sleep 0.1

    # Calculate progress
    progress=$((i * 100 / total_items))

    # Display progress bar
    printf "\rProgress: [%-50s] %d%%" $(printf "%.0s" $(seq 1 $((progress/2)))) $progress
done

echo
echo "Complete!"

```

Loops are powerful constructs that enable automation and batch processing in your shell scripts. Understanding when and how to use each type of loop will help you write more efficient and effective scripts.

Functions

Function Basics

Functions are reusable blocks of code that help organize your scripts, reduce repetition, and make your code more maintainable. They allow you to break complex scripts into smaller, manageable pieces.

What are Functions?

Functions in Bash are named blocks of code that can be called multiple times throughout your script. They help you:

- **Organize code:** Break large scripts into logical pieces
- **Reduce repetition:** Write once, use many times
- **Improve readability:** Make scripts easier to understand
- **Enable testing:** Test individual components
- **Facilitate maintenance:** Update code in one place

Function Syntax

Method 1: Using the function keyword

```
function function_name() {  
    # function body  
    commands  
}
```

Method 2: POSIX-compatible syntax (preferred)

```
function_name() {  
    # function body  
    commands  
}
```

Basic Function Examples

Simple Function

```
#!/bin/bash

# Define a function
greet() {
    echo "Hello, World!"
}

# Call the function
greet
```

Function with Commands

```
#!/bin/bash

show_date() {
    echo "Current date and time:"
    date
    echo "Uptime:"
    uptime
}

# Call the function
show_date
```

Function Parameters

Functions can accept parameters (arguments) to make them more flexible:

Accessing Parameters

```
#!/bin/bash

greet_user() {
    echo "Hello, $1!"
    echo "Welcome to $2"
}

# Call function with arguments
```

```
greet_user "Alice" "Bash scripting"
```

Multiple Parameters

```
#!/bin/bash

user_info() {
    local name="$1"
    local age="$2"
    local city="$3"

    echo "Name: $name"
    echo "Age: $age"
    echo "City: $city"
}

# Call with multiple arguments
user_info "John Doe" "30" "New York"
```

Parameter Variables

```
#!/bin/bash

show_params() {
    echo "Function name: $0"
    echo "First parameter: $1"
    echo "Second parameter: $2"
    echo "All parameters: $@"
    echo "Number of parameters: $#"
```

echo "All parameters as single string: \$*"
}

```
show_params "arg1" "arg2" "arg3"
```

Return Values

Functions can return values using the **return** statement:

Return Exit Status

```
#!/bin/bash

check_file() {
    local filename="$1"

    if [ -f "$filename" ]; then
        echo "File $filename exists"
        return 0 # Success
    else
        echo "File $filename does not exist"
        return 1 # Failure
    fi
}

# Use function and check return value
if check_file "test.txt"; then
    echo "File check passed"
else
    echo "File check failed"
fi
```

Return Values via Echo

```
#!/bin/bash

get_file_size() {
    local filename="$1"

    if [ -f "$filename" ]; then
        stat -f%z "$filename" 2>/dev/null || stat -c%s "$filename" 2>/dev/null
    else
        echo "0"
    fi
}

# Capture function output
file_size=$(get_file_size "test.txt")
echo "File size: $file_size bytes"
```

Local Variables

Use `local` to create variables that exist only within the function:

Without Local Variables (Global)

```
#!/bin/bash

global_example() {
    name="Alice" # This modifies global variable
    echo "Inside function: $name"
}

name="Bob"
echo "Before function: $name"
global_example
echo "After function: $name" # Changed to Alice
```

With Local Variables

```
#!/bin/bash

local_example() {
    local name="Alice" # This is local to the function
    echo "Inside function: $name"
}

name="Bob"
echo "Before function: $name"
local_example
echo "After function: $name" # Still Bob
```

Best Practice Example

```
#!/bin/bash

calculate_area() {
    local length="$1"
    local width="$2"
    local area

    area=$((length * width))
    echo "$area"
}

# Use the function
room_area=$(calculate_area 10 12)
```



```
echo "Room area: $room_area square feet"
```

Practical Function Examples

File Operations Function

```
#!/bin/bash

backup_file() {
    local source_file="$1"
    local backup_dir="${2:-./backup}"

    # Validate input
    if [ -z "$source_file" ]; then
        echo "Error: Source file not specified"
        return 1
    fi

    if [ ! -f "$source_file" ]; then
        echo "Error: Source file '$source_file' does not exist"
        return 1
    fi

    # Create backup directory if it doesn't exist
    mkdir -p "$backup_dir"

    # Create backup with timestamp
    local backup_name="$(basename "$source_file").backup.$(date +%Y%m%d_%H%M%S)"

    if cp "$source_file" "$backup_dir/$backup_name"; then
        echo "Backup created: $backup_dir/$backup_name"
        return 0
    else
        echo "Error: Failed to create backup"
        return 1
    fi
}

# Use the function
backup_file "/etc/hosts"
backup_file "important.txt" "/safe/location"
```

System Information Function

```
#!/bin/bash

show_system_info() {
    local info_type="$1"

    case "$info_type" in
        "cpu")
            echo "CPU Information:"
            lscpu | grep "Model name" || echo "CPU info not available"
            ;;
        "memory")
            echo "Memory Information:"
            free -h
            ;;
        "disk")
            echo "Disk Usage:"
            df -h
            ;;
        "all")
            show_system_info "cpu"
            echo
            show_system_info "memory"
            echo
            show_system_info "disk"
            ;;
        *)
            echo "Usage: show_system_info {cpu|memory|disk|all}"
            return 1
            ;;
    esac
}

# Use the function
show_system_info "all"
```

String Processing Function

```
#!/bin/bash

process_string() {
    local input="$1"
    local operation="$2"
```

```

    case "$operation" in
        "upper")
            echo "${input^^}"
            ;;
        "lower")
            echo "${input,,}"
            ;;
        "length")
            echo "${#input}"
            ;;
        "reverse")
            echo "$input" | rev
            ;;
        "words")
            echo "$input" | wc -w
            ;;
        *)
            echo "Usage: process_string <string> {upper|lower|length|reverse|words}"
            return 1
            ;;
    esac
}

# Use the function
text="Hello World"
echo "Original: $text"
echo "Uppercase: $(process_string "$text" "upper")"
echo "Lowercase: $(process_string "$text" "lower")"
echo "Length: $(process_string "$text" "length")"
echo "Reversed: $(process_string "$text" "reverse")"
echo "Word count: $(process_string "$text" "words")"

```

Function Libraries

You can create reusable function libraries:

Create a Library File (utils.sh)

```

#!/bin/bash
# utils.sh - Utility functions library

# Logging function
log() {
    local level="$1"

```

```

    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message"
}

# Check if command exists
command_exists() {
    command -v "$1" >/dev/null 2>&1
}

# Validate email format
is_valid_email() {
    local email="$1"
    [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
}

# Get file extension
get_extension() {
    local filename="$1"
    echo "${filename##*.}"
}

# Create directory if it doesn't exist
ensure_directory() {
    local dir="$1"
    if [ ! -d "$dir" ]; then
        mkdir -p "$dir"
        log "INFO" "Created directory: $dir"
    fi
}

```

Use the Library in Your Script

```

#!/bin/bash
# main_script.sh

# Source the utility functions
source ./utils.sh

# Use the functions
log "INFO" "Script started"

if command_exists "git"; then
    log "INFO" "Git is available"
else

```

```

        log "ERROR" "Git is not installed"
        exit 1
    fi

    email="user@example.com"
    if is_valid_email "$email"; then
        log "INFO" "Email format is valid"
    else
        log "ERROR" "Invalid email format"
    fi

    filename="document.pdf"
    extension=$(get_extension "$filename")
    log "INFO" "File extension: $extension"

    ensure_directory "/tmp/myapp"
    log "INFO" "Script completed"

```

Function Best Practices

1. Use Descriptive Names

```

# Good
calculate_monthly_payment() {
    # function body
}

# Bad
calc() {
    # function body
}

```

2. Validate Input Parameters

```

backup_database() {
    local db_name="$1"
    local backup_path="$2"

    # Validate required parameters
    if [ -z "$db_name" ]; then
        echo "Error: Database name is required"
        return 1
    fi
}

```

```

    if [ -z "$backup_path" ]; then
        echo "Error: Backup path is required"
        return 1
    fi

    # Function logic here
}

```

3. Use Local Variables

```

process_data() {
    local input_file="$1"
    local output_file="$2"
    local temp_file
    local line_count

    # All variables are local to this function
    temp_file=$(mktemp)
    line_count=$(wc -l < "$input_file")

    # Process data...
}

```

4. Provide Usage Information

```

convert_temperature() {
    local temp="$1"
    local from_unit="$2"
    local to_unit="$3"

    if [ $# -ne 3 ]; then
        echo "Usage: convert_temperature <temperature> <from_unit> <to_unit>"
        echo "Units: C (Celsius), F (Fahrenheit), K (Kelvin)"
        return 1
    fi

    # Conversion logic here
}

```

5. Handle Errors Gracefully

```
download_file() {
    local url="$1"
    local destination="$2"

    if ! command -v curl >/dev/null 2>&1; then
        echo "Error: curl is not installed"
        return 1
    fi

    if ! curl -o "$destination" "$url"; then
        echo "Error: Failed to download $url"
        return 1
    fi

    echo "Successfully downloaded to $destination"
    return 0
}
```

Common Function Patterns

Configuration Function

```
load_config() {
    local config_file="${1:-config.conf}"

    if [ -f "$config_file" ]; then
        source "$config_file"
        echo "Configuration loaded from $config_file"
    else
        echo "Warning: Configuration file $config_file not found"
        echo "Using default settings"
    fi
}
```

Cleanup Function

```
cleanup() {
    local temp_files=("$@")

    echo "Cleaning up temporary files..."
    for file in "${temp_files[@]}; do
```

```

        if [ -f "$file" ]; then
            rm "$file"
            echo "Removed: $file"
        fi
    done
}

# Set trap to call cleanup on exit
trap 'cleanup "$temp_file1" "$temp_file2"' EXIT

```

Menu Function

```

show_menu() {
    echo "=== Main Menu ==="
    echo "1. Option 1"
    echo "2. Option 2"
    echo "3. Option 3"
    echo "4. Exit"
    echo -n "Choose an option: "
}

handle_choice() {
    local choice="$1"

    case "$choice" in
        1) echo "You chose option 1" ;;
        2) echo "You chose option 2" ;;
        3) echo "You chose option 3" ;;
        4) echo "Goodbye!"; exit 0 ;;
        *) echo "Invalid choice" ;;
    esac
}

```

Functions are powerful tools that make your Bash scripts more organized, maintainable, and reusable. They're essential for writing professional-quality shell scripts that are easy to understand and modify.

Misc

Loop from 1 to 5

Loops are one of the core building blocks in Bash scripting. They let you run a set of commands repeatedly, which is useful for anything from processing files in a directory to automating system tasks. Among the different types of loops, the for loop is one of the most commonly used. It lets you iterate over a sequence of values, such as numbers, strings, or items in an array.

In this guide, you'll learn how Bash for loops work. We'll go over the standard syntax, the C-style variant, and how to use `break` and `continue` to control loop execution. You'll also see how nested loops work, with plenty of examples along the way to show how you can use them in real scripts.

Standard Bash for Loop

The standard Bash for loop has the following syntax:

```
for variable in list do commands done
```

Here, `variable` is a temporary variable that takes on each value from the list in turn, and the commands are executed for each iteration. The list can be a list of values separated by spaces, strings separated by spaces, a range of numbers, or even the output of a command.

Loop Over Number Ranges

One common use case for the for loop is to iterate over a range of numbers. This can be achieved using the `seq` command, which generates a sequence of numbers:

```
for i in $(seq 1 5) do echo $i done
```

Here's how it works:

The loop starts by assigning the first value from the sequence `$(seq 1 5)` to the variable `i`. The `echo $i` command within the loop body is then executed. The loop then moves to the next value in the sequence, assigns it to the variable, and executes the `echo $i` command again. This process continues until all values in the sequence have been iterated over. You can also specify an increment value for the sequence:

Loop from 1 to 10, incrementing by 2

`for i in $(seq 1 2 10) do echo $i done` Here is another common alternative syntax for specifying a range of numbers:

```
for i in {1..5}
do
    echo "Iteration $i"
done
```

The above syntax is more common in bash scripts. Starting from Bash 4, it's now possible to specify an increment when using ranges. Here is what it looks like:

```
# start..end..increment
for i in {1..10..2}
do
    echo "Iteration $i"
done
```

Loop Over Strings

The for loop can also iterate over a list of strings. Consider the following example:

Loop over a list of strings

```
for color in red green blue
do
    echo "The color is $color"
done
```

In this case, the loop iterates over the specified color strings, displaying each one in the output.

You can also use the for loop to iterate over characters in a string:

```
mystring="Hello, World!"
for char in $(echo "$mystring" | grep -o .)
do
    echo "$char"
done
```

In the above example:

The string “Hello, World!” is stored in the variable mystring. The `grep -o .` command matches and prints each character in the string individually. The for loop iterates over each character output by `grep`, assigning it to the variable `char`. Within the loop body, the current character is printed using the `echo` command. Loop Over Arrays

Bash arrays provide a convenient way to store multiple values. Let’s see how to use a for loop to iterate over the elements of an array:

Define an array

```
colors=("red" "green" "blue")
```

Loop over the array

```
for color in "${colors[@]}"
do
    echo "The color is $color"
done
```

Here we defined an array named `colors` and iterate over each element of the array echoing its value. Note the use of the `[@]` syntax to expand the array into a list of values.

If my Linux guides, cheat sheets, or deep dives have helped you learn or solve a problem, consider going paid. It keeps this work going and independent.

Upgrade to paid C-Style Bash for Loop

Bash also supports a C-style for loop, which provides more control over loop variables and conditions. The syntax for the C-style for loop is:

`for (( variable assignment; condition; iteration process )) do commands done` Here, variable assignment is executed once before the loop starts, condition is the condition that is evaluated before each iteration, and iteration process is executed after each iteration.

The following example prints numbers from 1 to 5.

Loop from 1 to 5

`for ((i=1; i<=5; i++)) do echo $i done` There are a couple of things you need to know about this syntax:

Variable value assignments can contain spaces. It's possible to omit the dollar sign for the variable in the condition. The equation for the iteration process doesn't use the `expr` command format to increment the value of the value. Break and Continue Statements

The `break` and `continue` statements can be used to control the flow of execution within a `for` loop, allowing you to skip or terminate iterations based on specific conditions. These statements can be particularly useful when dealing with complex or nested loops, or when you want to implement specific logic within your loop.

Break Statement

The `break` statement terminates the loop immediately and transfers control to the next statement following the loop. This can be useful in situations where you want to exit the loop early based on a certain condition, such as when you've found the desired value or when a specific condition is met.

Loop through an array and break when a specific value is found

```
fruits=("apple" "banana" "orange" "grape" "pear")

for fruit in "${fruits[@]}"
do
    if [ "$fruit" == "orange" ]
    then
        echo "Found orange!"
        break
    fi
    echo "Current fruit: $fruit"
done
```

In this example, the loop iterates over the fruits array, and when the value “orange” is encountered, the break statement is executed, terminating the loop and printing “Found orange!”.

Continue Statement

The continue statement skips the current iteration of the loop and moves to the next iteration. This can be useful when you want to skip certain iterations based on a specific condition, such as when dealing with invalid or unwanted data.

Loop through a range of numbers and skip multiples of 3

`for i in $(seq 1 10) do if [  $((i \% 3))$  -eq 0 ] then continue fi echo $i done` In this example, the loop iterates over the numbers from 1 to 10, but it skips the multiples of 3 using the `continue` statement.

Both `break` and `continue` statements can be used in combination with conditional statements (`if`, `case`, etc.) to implement complex logic within your loops. However, it's important to use them judiciously and avoid excessive use, as this can make your code harder to read and maintain.

Nested For Loops

In Bash, it's possible to have one `for` loop nested inside another `for` loop. This construct is known as a nested loop, and it allows you to iterate over multidimensional data structures or perform operations that require nested iterations.

The syntax for a nested `for` loop is as follows:

`for outer_variable in outer_list do for inner_variable in inner_list do commands done done` Here, the outer loop iterates over `outer_list`, and for each iteration of the outer loop, the inner loop iterates over the `inner_list`. The commands are executed for each combination of the outer and inner loop variables.

Consider the following example:

`for i in $(seq 1 2) do for j in $(seq 1 3) do echo "Outer: $i, Inner: $j" done done` Let's break down the code:

The outer loop iterates over the numbers 1 and 2, assigning each value to the variable `i`. For each iteration of the outer loop, the inner loop iterates over the numbers 1, 2, and 3, assigning each value to the variable `j`. Within the inner loop body, the current values of `i` and `j` are printed using the `echo` command. Thanks for reading!

If you enjoyed this content, don't forget to leave a like and subscribe to get more posts like this every week.

Workbook

Basic Bash Examples

This section contains fundamental Bash scripting examples to help you practice and understand core concepts.

Hello World Variations

Simple Hello World

```
#!/bin/bash
echo "Hello, World!"
```

Hello World with Variables

```
#!/bin/bash
name="World"
echo "Hello, $name!"
```

Interactive Hello World

```
#!/bin/bash
echo "What's your name?"
read name
echo "Hello, $name! Nice to meet you."
```

Hello World with Date

```
#!/bin/bash
echo "Hello, World!"
echo "Today is: $(date)"
echo "Current user: $(whoami)"
```

Variable Examples

Basic Variable Operations

```
#!/bin/bash
# Variable assignment
first_name="John"
last_name="Doe"
age=25

# Variable usage
echo "Name: $first_name $last_name"
echo "Age: $age"

# Variable concatenation
full_name="$first_name $last_name"
echo "Full name: $full_name"
```

Environment Variables

```
#!/bin/bash
echo "Your home directory: $HOME"
echo "Your username: $USER"
echo "Your shell: $SHELL"
echo "Current path: $PATH"
```

Variable Default Values

```
#!/bin/bash
# Set default values
database_host=${DB_HOST:-"localhost"}
database_port=${DB_PORT:-"5432"}
database_name=${DB_NAME:-"myapp"}

echo "Database configuration:"
echo "Host: $database_host"
echo "Port: $database_port"
echo "Database: $database_name"
```

Variable Length and Manipulation

```
#!/bin/bash
text="Hello World"

echo "Original text: $text"
echo "Length: ${#text}"
echo "Uppercase: ${text^^}"
echo "Lowercase: ${text,,}"
echo "First 5 characters: ${text:0:5}"
echo "Last 5 characters: ${text: -5}"
```

Command Line Arguments

Basic Argument Handling

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Argument Validation

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <name> [age]"
    exit 1
fi

name="$1"
age="${2:-unknown}"

echo "Name: $name"
echo "Age: $age"
```

Processing Multiple Arguments

```
#!/bin/bash
echo "Processing all arguments:"
for arg in "$@"; do
    echo "Argument: $arg"
done
```

Simple Calculations

Basic Arithmetic

```
#!/bin/bash
a=10
b=5

echo "a = $a, b = $b"
echo "Addition: $((a + b))"
echo "Subtraction: $((a - b))"
echo "Multiplication: $((a * b))"
echo "Division: $((a / b))"
echo "Modulo: $((a % b))"
```

Calculator Script

```
#!/bin/bash
echo "Simple Calculator"
read -p "Enter first number: " num1
read -p "Enter operator (+, -, *, /): " op
read -p "Enter second number: " num2

case $op in
    +) result=$((num1 + num2)) ;;
    -) result=$((num1 - num2)) ;;
    \*) result=$((num1 * num2)) ;;
    /)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
            exit 1
        fi
        result=$((num1 / num2))
        ;;
    *) echo "Invalid operator"; exit 1 ;;
```

```

esac

echo "Result: $num1 $op $num2 = $result"

```

Temperature Converter

```

#!/bin/bash
echo "Temperature Converter"
echo "1. Celsius to Fahrenheit"
echo "2. Fahrenheit to Celsius"
read -p "Choose option (1 or 2): " choice

case $choice in
    1)
        read -p "Enter temperature in Celsius: " celsius
        fahrenheit=$(echo "scale=2; $celsius * 9/5 + 32" | bc)
        echo "$celsius°C = $fahrenheit°F"
        ;;
    2)
        read -p "Enter temperature in Fahrenheit: " fahrenheit
        celsius=$(echo "scale=2; ($fahrenheit - 32) * 5/9" | bc)
        echo "$fahrenheit°F = $celsius°C"
        ;;
    *)
        echo "Invalid choice"
        exit 1
        ;;
esac

```

File Operations

File Information

```

#!/bin/bash
filename="$1"

if [ -z "$filename" ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

if [ -f "$filename" ]; then
    echo "File: $filename"

```

```

    echo "Size: $(stat -c%s "$filename") bytes"
    echo "Last modified: $(stat -c%y "$filename")"
    echo "Permissions: $(stat -c%A "$filename")"
    echo "Owner: $(stat -c%U "$filename")"
    echo "Group: $(stat -c%G "$filename")"
else
    echo "File '$filename' does not exist"
fi

```

File Backup

```

#!/bin/bash
source_file="$1"

if [ -z "$source_file" ]; then
    echo "Usage: $0 <source_file>"
    exit 1
fi

if [ ! -f "$source_file" ]; then
    echo "Error: File '$source_file' does not exist"
    exit 1
fi

backup_file="${source_file}.backup.$(date +%Y%m%d_%H%M%S)"

if cp "$source_file" "$backup_file"; then
    echo "Backup created: $backup_file"
else
    echo "Error: Failed to create backup"
    exit 1
fi

```

Directory Listing

```

#!/bin/bash
directory="${1:-.}"

echo "Contents of directory: $directory"
echo "======"

if [ -d "$directory" ]; then
    for item in "$directory"/*; do
        if [ -f "$item" ]; then

```



```

        echo "FILE: $(basename "$item")"
    elif [ -d "$item" ]; then
        echo "DIR: $(basename "$item")"
    fi
done
else
    echo "Error: '$directory' is not a directory"
fi

```

Text Processing

Word Count

```

#!/bin/bash
filename="$1"

if [ -z "$filename" ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

if [ -f "$filename" ]; then
    lines=$(wc -l < "$filename")
    words=$(wc -w < "$filename")
    chars=$(wc -c < "$filename")

    echo "File: $filename"
    echo "Lines: $lines"
    echo "Words: $words"
    echo "Characters: $chars"
else
    echo "File '$filename' does not exist"
fi

```

Text Search

```

#!/bin/bash
if [ $# -ne 2 ]; then
    echo "Usage: $0 <search_term> <filename>"
    exit 1
fi

search_term="$1"

```

```

filename="$2"

if [ ! -f "$filename" ]; then
    echo "File '$filename' does not exist"
    exit 1
fi

echo "Searching for '$search_term' in '$filename':"
grep -n "$search_term" "$filename"

count=$(grep -c "$search_term" "$filename")
echo "Found $count occurrences"

```

Line Numbering

```

#!/bin/bash
filename="$1"

if [ -z "$filename" ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

if [ -f "$filename" ]; then
    line_number=1
    while IFS= read -r line; do
        printf "%3d: %s\n" $line_number "$line"
        ((line_number++))
    done < "$filename"
else
    echo "File '$filename' does not exist"
fi

```

System Information

System Status

```

#!/bin/bash
echo "=== System Information ==="
echo "Hostname: $(hostname)"
echo "Uptime: $(uptime -p)"
echo "Current user: $(whoami)"
echo "Current directory: $(pwd)"

```

```
echo "Date: $(date)"
echo

echo "=== Memory Usage ==="
free -h

echo
echo "=== Disk Usage ==="
df -h

echo
echo "=== CPU Information ==="
lscpu | grep "Model name"
```

Process Monitor

```
#!/bin/bash
echo "Top 10 processes by CPU usage:"
ps aux --sort=-%cpu | head -11

echo
echo "Top 10 processes by memory usage:"
ps aux --sort=-%mem | head -11
```

Network Information

```
#!/bin/bash
echo "=== Network Information ==="
echo "Hostname: $(hostname)"
echo "Domain: $(hostname -d 2>/dev/null || echo "Not set")"

echo
echo "=== Network Interfaces ==="
ip addr show | grep -E "[0-9]+:|inet "

echo
echo "=== Routing Table ==="
ip route show
```

User Input Examples

Menu System

```
#!/bin/bash
while true; do
    echo
    echo "=== Main Menu ==="
    echo "1. Show date and time"
    echo "2. Show current directory"
    echo "3. List files"
    echo "4. Show disk usage"
    echo "5. Exit"
    echo
    read -p "Choose an option (1-5): " choice

    case $choice in
        1) date ;;
        2) pwd ;;
        3) ls -la ;;
        4) df -h ;;
        5) echo "Goodbye!"; exit 0 ;;
        *) echo "Invalid choice. Please try again." ;;
    esac

    read -p "Press Enter to continue..."
done
```

User Registration

```
#!/bin/bash
echo "User Registration Form"
echo "===== "

read -p "First Name: " first_name
read -p "Last Name: " last_name
read -p "Email: " email
read -p "Age: " age
read -s -p "Password: " password
echo

echo
echo "Registration Summary:"
echo "Name: $first_name $last_name"
echo "Email: $email"
```

```

echo "Age: $age"
echo "Password: [hidden]"

read -p "Is this information correct? (y/n): " confirm
if [ "$confirm" = "y" ] || [ "$confirm" = "Y" ]; then
    echo "Registration completed!"
else
    echo "Registration cancelled."
fi

```

Quiz Game

```

#!/bin/bash
score=0
total_questions=3

echo "Welcome to the Bash Quiz!"
echo "===== "

# Question 1
echo "Question 1: What command shows the current directory?"
echo "a) pwd"
echo "b) cd"
echo "c) ls"
read -p "Your answer: " answer1

if [ "$answer1" = "a" ] || [ "$answer1" = "A" ]; then
    echo "Correct!"
    ((score++))
else
    echo "Wrong! The correct answer is 'a) pwd'"
fi

# Question 2
echo
echo "Question 2: What symbol is used for comments in bash?"
echo "a) //"
echo "b) #"
echo "c) /*"
read -p "Your answer: " answer2

if [ "$answer2" = "b" ] || [ "$answer2" = "B" ]; then
    echo "Correct!"
    ((score++))
else

```

```

        echo "Wrong! The correct answer is 'b) #' "
    fi

# Question 3
echo
echo "Question 3: What command lists files and directories?"
echo "a) list"
echo "b) dir"
echo "c) ls"
read -p "Your answer: " answer3

if [ "$answer3" = "c" ] || [ "$answer3" = "C" ]; then
    echo "Correct!"
    ((score++))
else
    echo "Wrong! The correct answer is 'c) ls'"
fi

echo
echo "Quiz completed!"
echo "Your score: $score out of $total_questions"

percentage=$((score * 100 / total_questions))
if [ $percentage -ge 80 ]; then
    echo "Excellent work!"
elif [ $percentage -ge 60 ]; then
    echo "Good job!"
else
    echo "Keep practicing!"
fi

```

String Manipulation

String Operations

```

#!/bin/bash
text="Hello World Programming"

echo "Original: $text"
echo "Length: ${#text}"
echo "Uppercase: ${text^^}"
echo "Lowercase: ${text,,}"
echo "First word: ${text%% *}"
echo "Last word: ${text##*}"
echo "Remove 'Hello': ${text/Hello/}"

```

```

echo "Replace 'World' with 'Bash': ${text/World/Bash}"
echo "First 10 chars: ${text:0:10}"
echo "From position 6: ${text:6}"

```

Palindrome Checker

```

#!/bin/bash
read -p "Enter a word to check if it's a palindrome: " word

# Convert to lowercase and remove spaces
clean_word=$(echo "$word" | tr '[:upper:]' '[:lower:]' | tr -d ' ')

# Reverse the string
reversed=$(echo "$clean_word" | rev)

if [ "$clean_word" = "$reversed" ]; then
    echo "'$word' is a palindrome!"
else
    echo "'$word' is not a palindrome."
fi

```

Password Generator

```

#!/bin/bash
read -p "Enter password length (default 12): " length
length=${length:-12}

# Generate password using /dev/urandom
password=$(tr -dc 'A-Za-z0-9!@#%&* ' < /dev/urandom | head -c "$length")

echo "Generated password: $password"
echo "Password strength:"
echo "Length: $length characters"

# Check password strength
if [[ $password =~ [A-Z] ]]; then
    echo "Contains uppercase letters"
fi

if [[ $password =~ [a-z] ]]; then
    echo "Contains lowercase letters"
fi

if [[ $password =~ [0-9] ]]; then

```

```

        echo " Contains numbers"
    fi

    if [[ $password =~ [!@#\$%\^&*] ]]; then
        echo " Contains special characters"
    fi

```

Array Examples

Basic Array Operations

```

#!/bin/bash
# Create array
fruits=("apple" "banana" "orange" "grape")

echo "All fruits: ${fruits[@]}"
echo "Number of fruits: ${#fruits[@]}"
echo "First fruit: ${fruits[0]}"
echo "Last fruit: ${fruits[-1]}"

# Add element
fruits+=("mango")
echo "After adding mango: ${fruits[@]}"

# Loop through array
echo "Fruits list:"
for i in "${!fruits[@]}"; do
    echo "$((i+1)). ${fruits[i]}"
done

```

Shopping List

```

#!/bin/bash
declare -a shopping_list

echo "Shopping List Manager"
echo "===== "

while true; do
    echo
    echo "1. Add item"
    echo "2. Remove item"
    echo "3. Show list"

```



```

echo "4. Clear list"
echo "5. Exit"
read -p "Choose option: " choice

case $choice in
    1)
        read -p "Enter item to add: " item
        shopping_list+=("$item")
        echo "Added '$item' to the list"
        ;;
    2)
        if [ ${#shopping_list[@]} -eq 0 ]; then
            echo "List is empty"
        else
            echo "Current items:"
            for i in "${!shopping_list[@]}"; do
                echo "${(i+1)}. ${shopping_list[i]}"
            done
            read -p "Enter item number to remove: " num
            if [ "$num" -ge 1 ] && [ "$num" -le ${#shopping_list[@]} ]; then
                removed_item="${shopping_list[${(num-1)}]}"
                unset shopping_list[${(num-1)}]
                shopping_list=("${shopping_list[@]}") # Re-index array
                echo "Removed '$removed_item'"
            else
                echo "Invalid item number"
            fi
        fi
        ;;
    3)
        if [ ${#shopping_list[@]} -eq 0 ]; then
            echo "Shopping list is empty"
        else
            echo "Shopping List:"
            for i in "${!shopping_list[@]}"; do
                echo "${(i+1)}. ${shopping_list[i]}"
            done
        fi
        ;;
    4)
        shopping_list=()
        echo "List cleared"
        ;;
    5)
        echo "Goodbye!"
        exit 0
        ;;
)

```

```

        *)
            echo "Invalid choice"
            ;;
    esac
done

```

Grade Calculator

```

#!/bin/bash
declare -a grades
declare -a subjects

echo "Grade Calculator"
echo "======"

read -p "How many subjects? " num_subjects

for ((i=1; i<=num_subjects; i++)); do
    read -p "Enter subject $i name: " subject
    read -p "Enter grade for $subject: " grade

    subjects+=("$subject")
    grades+=("$grade")
done

# Calculate average
total=0
for grade in "${grades[@]}"; do
    total=$((total + grade))
done

average=$((total / num_subjects))

echo
echo "Grade Report:"
echo "======"
for i in "${!subjects[@]}"; do
    echo "${subjects[i]}: ${grades[i]}"
done

echo
echo "Average: $average"

if [ $average -ge 90 ]; then
    echo "Grade: A (Excellent)"

```

```
elif [ $average -ge 80 ]; then
    echo "Grade: B (Good)"
elif [ $average -ge 70 ]; then
    echo "Grade: C (Average)"
elif [ $average -ge 60 ]; then
    echo "Grade: D (Below Average)"
else
    echo "Grade: F (Fail)"
fi
```

Intermediate Bash Examples

This section contains more advanced Bash scripting examples that demonstrate intermediate concepts and real-world applications.

Function Examples

Library of Utility Functions

```
#!/bin/bash
# utils.sh - Utility functions library

# Logging function with levels
log() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    case "$level" in
        ERROR) echo "[${timestamp}] ERROR: $message" >&2 ;;
        WARN)  echo "[${timestamp}] WARN:  $message" ;;
        INFO)  echo "[${timestamp}] INFO:  $message" ;;
        DEBUG) [ "$DEBUG" = "1" ] && echo "[${timestamp}] DEBUG: $message" ;;
    esac
}

# Check if command exists
command_exists() {
    command -v "$1" >/dev/null 2>&1
}

# Validate email format
validate_email() {
    local email="$1"
    [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
}

# Get file extension
```

```

get_extension() {
    local filename="$1"
    echo "${filename##*.*}"
}

# Convert bytes to human readable format
bytes_to_human() {
    local bytes="$1"
    local units=("B" "KB" "MB" "GB" "TB")
    local unit=0

    while [ $bytes -gt 1024 ] && [ $unit -lt 4 ]; do
        bytes=$((bytes / 1024))
        ((unit++))
    done

    echo "$bytes ${units[$unit]}"
}

# Create backup with timestamp
backup_file() {
    local file="$1"
    local backup_dir="${2:-./backups}"

    if [ ! -f "$file" ]; then
        log ERROR "File '$file' does not exist"
        return 1
    fi

    mkdir -p "$backup_dir"
    local backup_name="$(basename "$file").backup.$(date +%Y%m%d_%H%M%S)"

    if cp "$file" "$backup_dir/$backup_name"; then
        log INFO "Backup created: $backup_dir/$backup_name"
        echo "$backup_dir/$backup_name"
        return 0
    else
        log ERROR "Failed to create backup"
        return 1
    fi
}

# Example usage
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    # Test the functions
    log INFO "Testing utility functions"

```

```

if command_exists "git"; then
    log INFO "Git is available"
else
    log WARN "Git is not installed"
fi

if validate_email "user@example.com"; then
    log INFO "Email format is valid"
else
    log ERROR "Invalid email format"
fi

echo "File extension of script.sh: $(get_extension "script.sh")"
echo "1048576 bytes = $(bytes_to_human 1048576)"
fi

```

Configuration Manager

```

#!/bin/bash
# config_manager.sh - Configuration file manager

CONFIG_FILE="app.conf"

# Default configuration
declare -A config=(
    [app_name]="MyApp"
    [version]="1.0.0"
    [debug]="false"
    [port]="8080"
    [database_host]="localhost"
    [database_port]="5432"
    [log_level]="INFO"
)

# Load configuration from file
load_config() {
    if [ -f "$CONFIG_FILE" ]; then
        while IFS='=' read -r key value; do
            # Skip comments and empty lines
            [[ $key =~ ^[[:space:]]*# ]] && continue
            [[ -z $key ]] && continue

            # Remove leading/trailing whitespace
            key=$(echo "$key" | xargs)
            value=$(echo "$value" | xargs)

```

```

        config["$key"]="$value"
    done < "$CONFIG_FILE"
    echo "Configuration loaded from $CONFIG_FILE"
else
    echo "Configuration file not found, using defaults"
fi
}

# Save configuration to file
save_config() {
    echo "# Application Configuration" > "$CONFIG_FILE"
    echo "# Generated on $(date)" >> "$CONFIG_FILE"
    echo >> "$CONFIG_FILE"

    for key in "${!config[@]}; do
        echo "$key=${config[$key]}" >> "$CONFIG_FILE"
    done

    echo "Configuration saved to $CONFIG_FILE"
}

# Get configuration value
get_config() {
    local key="$1"
    echo "${config[$key]}"
}

# Set configuration value
set_config() {
    local key="$1"
    local value="$2"
    config["$key"]="$value"
    echo "Set $key = $value"
}

# Show all configuration
show_config() {
    echo "Current Configuration:"
    echo "======"
    for key in "${!config[@]}; do
        printf "%-20s = %s\n" "$key" "${config[$key]}"
    done
}

# Interactive configuration editor
edit_config() {

```

```

while true; do
    echo
    echo "Configuration Editor"
    echo "======"
    echo "1. Show current config"
    echo "2. Edit a value"
    echo "3. Save config"
    echo "4. Load config"
    echo "5. Exit"

    read -p "Choose option: " choice

    case $choice in
        1) show_config ;;
        2)
            read -p "Enter key to edit: " key
            if [[ -n "${config[$key]}" ]]; then
                echo "Current value: ${config[$key]}"
                read -p "Enter new value: " value
                set_config "$key" "$value"
            else
                echo "Key '$key' not found"
            fi
            ;;
        3) save_config ;;
        4) load_config ;;
        5) break ;;
        *) echo "Invalid choice" ;;
    esac
done
}

# Main execution
load_config

if [ $# -eq 0 ]; then
    edit_config
else
    case "$1" in
        get) get_config "$2" ;;
        set) set_config "$2" "$3" ;;
        show) show_config ;;
        save) save_config ;;
        load) load_config ;;
        *) echo "Usage: $0 {get|set|show|save|load} [key] [value]" ;;
    esac
fi

```


Database Backup Script

```
#!/bin/bash
# db_backup.sh - Database backup script with rotation

# Configuration
DB_HOST="${DB_HOST:-localhost}"
DB_PORT="${DB_PORT:-5432}"
DB_USER="${DB_USER:-postgres}"
DB_NAME="${DB_NAME:-myapp}"
BACKUP_DIR="${BACKUP_DIR:-/backup/database}"
RETENTION_DAYS="${RETENTION_DAYS:-7}"
COMPRESSION="${COMPRESSION:-gzip}"

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*' | tee -a "$BACKUP_DIR/backup.log"
}

# Check dependencies
check_dependencies() {
    local missing_deps=()

    if ! command -v pg_dump >/dev/null 2>&1; then
        missing_deps+=("postgresql-client")
    fi

    if [ "$COMPRESSION" = "gzip" ] && ! command -v gzip >/dev/null 2>&1; then
        missing_deps+=("gzip")
    fi

    if [ ${#missing_deps[@]} -gt 0 ]; then
        log "ERROR: Missing dependencies: ${missing_deps[*]}"
        exit 1
    fi
}

# Create backup directory
setup_backup_dir() {
    if [ ! -d "$BACKUP_DIR" ]; then
        mkdir -p "$BACKUP_DIR"
        log "INFO: Created backup directory: $BACKUP_DIR"
    fi
}

# Perform database backup
```

```

backup_database() {
    local timestamp=$(date +%Y%m%d_%H%M%S)
    local backup_file="$BACKUP_DIR/${DB_NAME}_${timestamp}.sql"

    log "INFO: Starting backup of database '$DB_NAME'"

    # Set password if provided
    if [ -n "$DB_PASSWORD" ]; then
        export PGPASSWORD="$DB_PASSWORD"
    fi

    # Perform backup
    if pg_dump -h "$DB_HOST" -p "$DB_PORT" -U "$DB_USER" -d "$DB_NAME" > "$backup_file"; then
        log "INFO: Database backup completed: $backup_file"

        # Compress if requested
        if [ "$COMPRESSION" = "gzip" ]; then
            if gzip "$backup_file"; then
                backup_file="${backup_file}.gz"
                log "INFO: Backup compressed: $backup_file"
            else
                log "WARN: Compression failed"
            fi
        fi

        # Calculate file size
        local file_size=$(stat -c%s "$backup_file" 2>/dev/null || stat -f%z "$backup_file" 2>/dev/null)
        log "INFO: Backup size: $(numfmt --to=iec-i --suffix=B $file_size)"

        echo "$backup_file"
    else
        log "ERROR: Database backup failed"
        return 1
    fi
}

# Clean old backups
cleanup_old_backups() {
    log "INFO: Cleaning up backups older than $RETENTION_DAYS days"

    local deleted_count=0
    while IFS= read -r -d '' file; do
        rm "$file"
        log "INFO: Deleted old backup: $(basename "$file")"
        ((deleted_count++))
    done << (find "$BACKUP_DIR" -name "${DB_NAME}*.sql*" -mtime +$RETENTION_DAYS -print0)
}

```

```

    log "INFO: Deleted $deleted_count old backup(s)"
}

# Verify backup integrity
verify_backup() {
    local backup_file="$1"

    if [ ! -f "$backup_file" ]; then
        log "ERROR: Backup file not found: $backup_file"
        return 1
    fi

    # Check if file is compressed
    if [[ "$backup_file" == *.gz ]]; then
        if gzip -t "$backup_file"; then
            log "INFO: Backup file integrity verified (compressed)"
            return 0
        else
            log "ERROR: Backup file is corrupted (compressed)"
            return 1
        fi
    else
        # Check SQL file for basic structure
        if grep -q "PostgreSQL database dump" "$backup_file"; then
            log "INFO: Backup file integrity verified"
            return 0
        else
            log "ERROR: Backup file appears to be corrupted"
            return 1
        fi
    fi
}

# Send notification (email or webhook)
send_notification() {
    local status="$1"
    local message="$2"

    if [ -n "$NOTIFICATION_EMAIL" ] && command -v mail >/dev/null 2>&1; then
        echo "$message" | mail -s "Database Backup $status" "$NOTIFICATION_EMAIL"
        log "INFO: Notification sent to $NOTIFICATION_EMAIL"
    fi

    if [ -n "$WEBHOOK_URL" ] && command -v curl >/dev/null 2>&1; then
        curl -X POST -H "Content-Type: application/json" \
            -d "{\"text\": \"Database Backup $status: $message\"}" \
            "$WEBHOOK_URL" >/dev/null 2>&1
    fi
}

```

```

        log "INFO: Webhook notification sent"
    fi
}

# Main backup process
main() {
    log "INFO: Starting database backup process"

    check_dependencies
    setup_backup_dir

    if backup_file=$(backup_database); then
        if verify_backup "$backup_file"; then
            cleanup_old_backups
            send_notification "SUCCESS" "Database backup completed successfully: $(basename
            log "INFO: Backup process completed successfully"
        else
            send_notification "FAILED" "Backup verification failed"
            log "ERROR: Backup verification failed"
            exit 1
        fi
    else
        send_notification "FAILED" "Database backup failed"
        log "ERROR: Backup process failed"
        exit 1
    fi
}

# Handle command line arguments
case "${1:-backup}" in
    backup) main ;;
    verify) verify_backup "$2" ;;
    cleanup) cleanup_old_backups ;;
    *)
        echo "Usage: $0 {backup|verify|cleanup}"
        echo "Environment variables:"
        echo "  DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME"
        echo "  BACKUP_DIR, RETENTION_DAYS, COMPRESSION"
        echo "  NOTIFICATION_EMAIL, WEBHOOK_URL"
        exit 1
    ;;
esac

```

File Processing Examples

Log Analyzer

```
#!/bin/bash
# log_analyzer.sh - Analyze web server logs

LOG_FILE="${1:-/var/log/apache2/access.log}"
OUTPUT_DIR="${2:-./log_analysis}"

# Check if log file exists
if [ ! -f "$LOG_FILE" ]; then
    echo "Error: Log file '$LOG_FILE' not found"
    exit 1
fi

# Create output directory
mkdir -p "$OUTPUT_DIR"

echo "Analyzing log file: $LOG_FILE"
echo "Output directory: $OUTPUT_DIR"
echo "======"

# Basic statistics
echo "Generating basic statistics..."
total_requests=$(wc -l < "$LOG_FILE")
echo "Total requests: $total_requests" > "$OUTPUT_DIR/summary.txt"

# Unique IP addresses
unique_ips=$(awk '{print $1}' "$LOG_FILE" | sort -u | wc -l)
echo "Unique IP addresses: $unique_ips" >> "$OUTPUT_DIR/summary.txt"

# Date range
first_date=$(head -1 "$LOG_FILE" | awk '{print $4}' | tr -d '[')
last_date=$(tail -1 "$LOG_FILE" | awk '{print $4}' | tr -d '[')
echo "Date range: $first_date to $last_date" >> "$OUTPUT_DIR/summary.txt"

# Top 10 IP addresses
echo "Generating top IP addresses..."
awk '{print $1}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -10 > "$OUTPUT_DIR/top_ips.txt"

# Top 10 requested pages
echo "Generating top requested pages..."
awk '{print $7}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -10 > "$OUTPUT_DIR/top_pages.txt"

# HTTP status codes
```

```

echo "Analyzing HTTP status codes..."
awk '{print $9}' "$LOG_FILE" | sort | uniq -c | sort -nr > "$OUTPUT_DIR/status_codes.txt"

# User agents
echo "Analyzing user agents..."
awk -F'"' '{print $6}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -20 > "$OUTPUT_DIR/user_agents.txt"

# Hourly traffic distribution
echo "Generating hourly traffic distribution..."
awk '{print $4}' "$LOG_FILE" | cut -d: -f2 | sort | uniq -c > "$OUTPUT_DIR/hourly_traffic.txt"

# Error analysis (4xx and 5xx)
echo "Analyzing errors..."
awk '$9 ~ /^[45]/ {print $9, $7}' "$LOG_FILE" | sort | uniq -c | sort -nr > "$OUTPUT_DIR/errors.txt"

# Bandwidth usage (if log format includes bytes)
echo "Calculating bandwidth usage..."
if awk '{print $10}' "$LOG_FILE" | head -1 | grep -q '^[0-9]*$'; then
    total_bytes=$(awk '{sum += $10} END {print sum}' "$LOG_FILE")
    echo "Total bandwidth: $(numfmt --to=iec-i --suffix=B $total_bytes)" >> "$OUTPUT_DIR/summary.txt"
fi

# Generate HTML report
echo "Generating HTML report..."
cat > "$OUTPUT_DIR/report.html" << EOF
<!DOCTYPE html>
<html>
<head>
    <title>Log Analysis Report</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        .section { margin: 20px 0; }
        .data { background: #f5f5f5; padding: 10px; border-radius: 5px; }
        pre { white-space: pre-wrap; }
    </style>
</head>
<body>
    <h1>Log Analysis Report</h1>
    <p>Generated on: $(date)</p>

    <div class="section">
        <h2>Summary</h2>
        <div class="data">
            <pre>$(cat "$OUTPUT_DIR/summary.txt")</pre>
        </div>
    </div>

```

```

<div class="section">
  <h2>Top 10 IP Addresses</h2>
  <div class="data">
    <pre>$(cat "$OUTPUT_DIR/top_ips.txt")</pre>
  </div>
</div>

<div class="section">
  <h2>Top 10 Requested Pages</h2>
  <div class="data">
    <pre>$(cat "$OUTPUT_DIR/top_pages.txt")</pre>
  </div>
</div>

<div class="section">
  <h2>HTTP Status Codes</h2>
  <div class="data">
    <pre>$(cat "$OUTPUT_DIR/status_codes.txt")</pre>
  </div>
</div>
</body>
</html>
EOF

echo "Analysis complete! Results saved in: $OUTPUT_DIR"
echo "View the HTML report: $OUTPUT_DIR/report.html"

```

File Organizer

```

#!/bin/bash
# file_organizer.sh - Organize files by type and date

SOURCE_DIR="${1:-.}"
TARGET_DIR="${2:-./organized}"
DRY_RUN="${DRY_RUN:-false}"

# File type associations
declare -A file_types=(
  ["jpg,jpeg,png,gif,bmp,tiff"]="Images"
  ["mp4,avi,mkv,mov,wmv,flv"]="Videos"
  ["mp3,wav,flac,aac,ogg"]="Audio"
  ["pdf,doc,docx,txt,rtf,odt"]="Documents"
  ["zip,rar,7z,tar,gz,bz2"]="Archives"
  ["exe,msi,deb,rpm,dmg"]="Executables"
  ["html,css,js,php,py,java,cpp,c"]="Code"

```

```

)

# Logging function
log() {
    echo "$(date '+%H:%M:%S') - $*"
}

# Get file category based on extension
get_file_category() {
    local filename="$1"
    local extension="${filename##*.}"
    extension=$(echo "$extension" | tr '[:upper:]' '[:lower:]')

    for types in "${!file_types[@]}; do
        if [[ "$types," == "$extension,*" ]]; then
            echo "${file_types[$types]}"
            return
        fi
    done

    echo "Other"
}

# Create directory structure
create_directory() {
    local dir="$1"
    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN: Would create directory: $dir"
    else
        mkdir -p "$dir"
        log "Created directory: $dir"
    fi
}

# Move file
move_file() {
    local source="$1"
    local target="$2"

    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN: Would move '$source' to '$target'"
    else
        if mv "$source" "$target"; then
            log "Moved: $(basename "$source") -> $target"
        else
            log "ERROR: Failed to move $(basename "$source")"
        fi
    fi
}

```



```

    fi
}

# Organize files
organize_files() {
    local source_dir="$1"
    local target_dir="$2"
    local file_count=0
    local dir_count=0

    log "Starting file organization..."
    log "Source: $source_dir"
    log "Target: $target_dir"

    # Process all files in source directory
    find "$source_dir" -maxdepth 1 -type f | while read -r file; do
        # Skip hidden files unless specified
        if [[ "$(basename "$file")" == .* ]] && [ "$INCLUDE_HIDDEN" != "true" ]; then
            continue
        fi

        # Get file info
        filename=$(basename "$file")
        category=$(get_file_category "$filename")

        # Get file date
        if command -v stat >/dev/null 2>&1; then
            file_date=$(stat -c %Y "$file" 2>/dev/null || stat -f %m "$file" 2>/dev/null)
            year=$(date -d "@$file_date" +%Y 2>/dev/null || date -r "$file_date" +%Y 2>/dev/null)
            month=$(date -d "@$file_date" +%m 2>/dev/null || date -r "$file_date" +%m 2>/dev/null)
        else
            year=$(date +%Y)
            month=$(date +%m)
        fi

        # Create target directory structure
        target_path="$target_dir/$category/$year/$month"
        create_directory "$target_path"

        # Move file
        move_file "$file" "$target_path/$filename"
        ((file_count++))
    done

    log "Organization complete. Processed $file_count files."
}

```

```

# Generate report
generate_report() {
    local target_dir="$1"
    local report_file="$target_dir/organization_report.txt"

    log "Generating organization report..."

    {
        echo "File Organization Report"
        echo "======"
        echo "Generated on: $(date)"
        echo "Target directory: $target_dir"
        echo

        echo "Directory Structure:"
        echo "======"
        if [ -d "$target_dir" ]; then
            find "$target_dir" -type d | sort
        fi
        echo

        echo "File Count by Category:"
        echo "======"
        for category in "${file_types[@]}" "Other"; do
            if [ -d "$target_dir/$category" ]; then
                count=$(find "$target_dir/$category" -type f | wc -l)
                printf "%-15s: %d files\n" "$category" "$count"
            fi
        done

    } > "$report_file"

    log "Report saved: $report_file"
}

# Show usage
show_usage() {
    cat << EOF
Usage: $0 [source_dir] [target_dir]

Options:
    source_dir    Source directory to organize (default: current directory)
    target_dir    Target directory for organized files (default: ./organized)

Environment Variables:
    DRY_RUN=true      Show what would be done without actually moving files
    INCLUDE_HIDDEN=true Include hidden files in organization

```

Examples:

```
$0                                # Organize current directory
$0 /home/user/Downloads          # Organize Downloads folder
DRY_RUN=true $0                  # Preview organization without moving files
```

File Categories:

EOF

```
for types in "${!file_types[@]}"; do
    printf "  %-15s: %s\n" "${file_types[$types]}" "$types"
done
}

# Main execution
main() {
    if [ "$1" = "-h" ] || [ "$1" = "--help" ]; then
        show_usage
        exit 0
    fi

    if [ ! -d "$SOURCE_DIR" ]; then
        log "ERROR: Source directory '$SOURCE_DIR' does not exist"
        exit 1
    fi

    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN MODE - No files will be moved"
    fi

    organize_files "$SOURCE_DIR" "$TARGET_DIR"

    if [ "$DRY_RUN" != "true" ]; then
        generate_report "$TARGET_DIR"
    fi
}

main "$@"
```

CSV Processor

```
#!/bin/bash
# csv_processor.sh - Process and analyze CSV files

CSV_FILE="$1"
DELIMITER="${DELIMITER:-,}"
```

```

OUTPUT_FORMAT="${OUTPUT_FORMAT:-table}"

# Check if file exists
if [ ! -f "$CSV_FILE" ]; then
    echo "Usage: $0 <csv_file>"
    echo "Environment variables:"
    echo "  DELIMITER      - Field delimiter (default: ,)"
    echo "  OUTPUT_FORMAT - Output format: table, json, html (default: table)"
    exit 1
fi

# Function to get CSV headers
get_headers() {
    head -1 "$CSV_FILE" | tr "$DELIMITER" '\n' | nl -w2 -s': '
}

# Function to count rows
count_rows() {
    local total_rows=$(wc -l < "$CSV_FILE")
    local data_rows=$((total_rows - 1))
    echo "Total rows: $total_rows (including header)"
    echo "Data rows: $data_rows"
}

# Function to show column statistics
column_stats() {
    local column_num="$1"
    local column_name=$(head -1 "$CSV_FILE" | cut -d"$DELIMITER" -f"$column_num")

    echo "Statistics for column $column_num: $column_name"
    echo "===== "

    # Get column data (skip header)
    tail -n +2 "$CSV_FILE" | cut -d"$DELIMITER" -f"$column_num" > /tmp/column_data.tmp

    # Count non-empty values
    local non_empty=$(grep -v '^$' /tmp/column_data.tmp | wc -l)
    local empty=$(grep '^$' /tmp/column_data.tmp | wc -l)

    echo "Non-empty values: $non_empty"
    echo "Empty values: $empty"

    # Check if column contains numbers
    if grep -q '^[0-9]*\.[0-9]*$' /tmp/column_data.tmp; then
        echo "Data type: Numeric"

        # Calculate numeric statistics
    fi
}

```

```

    local sum=$(awk '{sum += $1} END {print sum}' /tmp/column_data.tmp)
    local count=$(grep -v '^$' /tmp/column_data.tmp | wc -l)
    local avg=$(echo "scale=2; $sum / $count" | bc -l 2>/dev/null || echo "N/A")
    local min=$(sort -n /tmp/column_data.tmp | head -1)
    local max=$(sort -n /tmp/column_data.tmp | tail -1)

    echo "Sum: $sum"
    echo "Average: $avg"
    echo "Min: $min"
    echo "Max: $max"
else
    echo "Data type: Text"

    # Show unique values count
    local unique_count=$(sort /tmp/column_data.tmp | uniq | wc -l)
    echo "Unique values: $unique_count"

    # Show most common values
    echo "Most common values:"
    sort /tmp/column_data.tmp | uniq -c | sort -nr | head -5
fi

rm -f /tmp/column_data.tmp
}

# Function to filter CSV data
filter_data() {
    local column="$1"
    local operator="$2"
    local value="$3"

    echo "Filtering: Column $column $operator '$value'"
    echo "======"

    # Show header
    head -1 "$CSV_FILE"

    # Apply filter based on operator
    case "$operator" in
        "="|"eq")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
            ;;
        "!="|"ne")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
            ;;
        ">"|"gt")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c

```

```

        ;;
        "<"|"lt")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" 'c
            ;;
        "contains")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" 'in
            ;;
        *)
            echo "Supported operators: =, !=, >, <, contains"
            return 1
            ;;
    esac
}

# Function to convert to JSON
to_json() {
    local headers=$(head -1 "$CSV_FILE" | tr "$DELIMITER" ' ')

    echo "["
    local first_row=true

    tail -n +2 "$CSV_FILE" | while IFS="$DELIMITER" read -r -a fields; do
        if [ "$first_row" = true ]; then
            first_row=false
        else
            echo -n ","
        fi

        echo -n " {"
        for i in "${!headers[@]}"; do
            if [ $i -gt 0 ]; then
                echo -n ", "
            fi
            echo -n "\"${headers[i]}\": \"${fields[i]}\","
        done
        echo -n "}"
    done

    echo
    echo "]"
}

# Function to convert to HTML table
to_html() {
    echo "<table border='1'>"
    echo "<thead><tr>"

```

```

# Headers
head -1 "$CSV_FILE" | tr "$DELIMITER" '\n' | while read -r header; do
    echo "<th>$header</th>"
done

echo "</tr></thead><tbody>"

# Data rows
tail -n +2 "$CSV_FILE" | while IFS="$DELIMITER" read -r -a fields; do
    echo "<tr>"
    for field in "${fields[@]}"; do
        echo "<td>$field</td>"
    done
    echo "</tr>"
done

echo "</tbody></table>"
}

# Function to show menu
show_menu() {
    echo
    echo "CSV Processor Menu"
    echo "======"
    echo "1. Show headers"
    echo "2. Count rows"
    echo "3. Column statistics"
    echo "4. Filter data"
    echo "5. Convert to JSON"
    echo "6. Convert to HTML"
    echo "7. Show first 10 rows"
    echo "8. Show last 10 rows"
    echo "9. Exit"
    echo
}

# Interactive mode
interactive_mode() {
    while true; do
        show_menu
        read -p "Choose option: " choice

        case $choice in
            1)
                echo "CSV Headers:"
                get_headers
                ;;

```

```

2)
    count_rows
    ;;
3)
    echo "Available columns:"
    get_headers
    read -p "Enter column number: " col_num
    column_stats "$col_num"
    ;;
4)
    echo "Available columns:"
    get_headers
    read -p "Enter column number: " col_num
    read -p "Enter operator (=, !=, >, <, contains): " operator
    read -p "Enter value: " value
    filter_data "$col_num" "$operator" "$value"
    ;;
5)
    to_json
    ;;
6)
    to_html
    ;;
7)
    echo "First 10 rows:"
    head -11 "$CSV_FILE" | column -t -s"$DELIMITER"
    ;;
8)
    echo "Last 10 rows:"
    tail -10 "$CSV_FILE" | column -t -s"$DELIMITER"
    ;;
9)
    echo "Goodbye!"
    exit 0
    ;;
*)
    echo "Invalid choice"
    ;;
esac

echo
read -p "Press Enter to continue..."
done
}

# Main execution

```



```

echo "Processing CSV file: $CSV_FILE"
echo "Delimiter: '$DELIMITER'"
echo

# If no additional arguments, start interactive mode
if [ $# -eq 1 ]; then
    interactive_mode
else
    # Command line mode
    case "$2" in
        headers) get_headers ;;
        count) count_rows ;;
        stats) column_stats "$3" ;;
        filter) filter_data "$3" "$4" "$5" ;;
        json) to_json ;;
        html) to_html ;;
        *)
            echo "Usage: $0 <csv_file> [command] [args...]"
            echo "Commands: headers, count, stats <col>, filter <col> <op> <val>, json, html"
            ;;
    esac
fi

```

System Administration Examples

System Monitor

```

#!/bin/bash
# system_monitor.sh - Comprehensive system monitoring script

ALERT_CPU_THRESHOLD=${ALERT_CPU_THRESHOLD:-80}
ALERT_MEMORY_THRESHOLD=${ALERT_MEMORY_THRESHOLD:-85}
ALERT_DISK_THRESHOLD=${ALERT_DISK_THRESHOLD:-90}
LOG_FILE=${LOG_FILE:-"/var/log/system_monitor.log"}
EMAIL_ALERT=${EMAIL_ALERT:-""}
WEBHOOK_URL=${WEBHOOK_URL:-""}

# Colors for output
RED='\033[0;31m'
YELLOW='\033[1;33m'
GREEN='\033[0;32m'
NC='\033[0m' # No Color

# Logging function

```

```

log_message() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    echo "[${timestamp}] [${level}] $message" >> "$LOG_FILE"

    case "$level" in
        ERROR) echo -e "${RED}${timestamp} ERROR: $message${NC}" ;;
        WARN)  echo -e "${YELLOW}${timestamp} WARN: $message${NC}" ;;
        INFO)  echo -e "${GREEN}${timestamp} INFO: $message${NC}" ;;
    esac
}

# Send alert notification
send_alert() {
    local subject="$1"
    local message="$2"

    # Email notification
    if [ -n "$EMAIL_ALERT" ] && command -v mail >/dev/null 2>&1; then
        echo "$message" | mail -s "$subject" "$EMAIL_ALERT"
        log_message INFO "Alert sent via email to $EMAIL_ALERT"
    fi

    # Webhook notification
    if [ -n "$WEBHOOK_URL" ] && command -v curl >/dev/null 2>&1; then
        curl -X POST -H "Content-Type: application/json" \
            -d "{\"text\":\"$subject: $message\"}" \
            "$WEBHOOK_URL" >/dev/null 2>&1
        log_message INFO "Alert sent via webhook"
    fi
}

# Check CPU usage
check_cpu() {
    local cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)
    cpu_usage=${cpu_usage%.*} # Remove decimal part

    echo "CPU Usage: ${cpu_usage}%"

    if [ "$cpu_usage" -gt "$ALERT_CPU_THRESHOLD" ]; then
        local message="High CPU usage detected: ${cpu_usage}% (threshold: ${ALERT_CPU_THRESH"
        log_message WARN "$message"
        send_alert "CPU Alert" "$message"
    fi
}

```

```

        # Show top processes
        echo "Top CPU consuming processes:"
        ps aux --sort=-%cpu | head -6
    else
        log_message INFO "CPU usage normal: ${cpu_usage}%"
    fi
}

# Check memory usage
check_memory() {
    local memory_info=$(free | grep Mem)
    local total=$(echo $memory_info | awk '{print $2}')
    local used=$(echo $memory_info | awk '{print $3}')
    local memory_usage=$((used * 100 / total))

    echo "Memory Usage: ${memory_usage}% ($(numfmt --to=iec $((used * 1024))) / $(numfmt --t

    if [ "$memory_usage" -gt "$ALERT_MEMORY_THRESHOLD" ]; then
        local message="High memory usage detected: ${memory_usage}% (threshold: ${ALERT_MEMO
        log_message WARN "$message"
        send_alert "Memory Alert" "$message"

        # Show top memory consuming processes
        echo "Top memory consuming processes:"
        ps aux --sort=-%mem | head -6
    else
        log_message INFO "Memory usage normal: ${memory_usage}%"
    fi
}

# Check disk usage
check_disk() {
    echo "Disk Usage:"
    df -h | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        usage=$(echo $output | awk '{print $5}' | cut -d '%' -f1)
        partition=$(echo $output | awk '{print $1}')
        mount_point=$(echo $output | awk '{print $6}')

        echo "  $mount_point: ${usage}%"

        if [ "$usage" -gt "$ALERT_DISK_THRESHOLD" ]; then
            local message="High disk usage detected on $mount_point: ${usage}% (threshold: $
            log_message WARN "$message"
            send_alert "Disk Alert" "$message"
        else
            log_message INFO "Disk usage normal on $mount_point: ${usage}%"
        fi
    done
}

```

```

done
}

# Check system load
check_load() {
    local load_avg=$(uptime | awk -F'load average:' '{print $2}')
    local load_1min=$(echo $load_avg | awk '{print $1}' | tr -d ',')
    local cpu_cores=$(nproc)

    echo "Load Average: $load_avg"
    echo "CPU Cores: $cpu_cores"

    # Check if 1-minute load is higher than number of cores
    if (( $(echo "$load_1min > $cpu_cores" | bc -l) )); then
        local message="High system load detected: $load_1min (cores: $cpu_cores)"
        log_message WARN "$message"
        send_alert "Load Alert" "$message"
    else
        log_message INFO "System load normal: $load_1min"
    fi
}

# Check running services
check_services() {
    local services=("ssh" "nginx" "apache2" "mysql" "postgresql")

    echo "Service Status:"
    for service in "${services[@]}; do
        if systemctl is-active --quiet "$service" 2>/dev/null; then
            echo "  $service: Running"
            log_message INFO "Service $service is running"
        elif systemctl list-unit-files | grep -q "^$service.service"; then
            echo "  $service: Stopped"
            log_message WARN "Service $service is stopped"
            send_alert "Service Alert" "Service $service is not running"
        fi
    done
}

# Check network connectivity
check_network() {
    local test_hosts=("8.8.8.8" "1.1.1.1" "google.com")

    echo "Network Connectivity:"
    for host in "${test_hosts[@]}; do
        if ping -c 1 -W 5 "$host" >/dev/null 2>&1; then
            echo "  $host: OK"
        fi
    done
}

```

```

        log_message INFO "Network connectivity to $host OK"
    else
        echo "  $host: FAILED"
        log_message ERROR "Network connectivity to $host failed"
        send_alert "Network Alert" "Cannot reach $host"
    fi
done
}

# Check log files for errors
check_logs() {
    local log_files=("/var/log/syslog" "/var/log/auth.log" "/var/log/kern.log")
    local time_threshold="1 hour ago"

    echo "Recent Log Errors:"
    for log_file in "${log_files[@]}; do
        if [ -f "$log_file" ]; then
            local error_count=$(find "$log_file" -newermt "$time_threshold" -exec grep -i "e
            echo "  $(basename "$log_file"): $error_count errors in last hour"

            if [ "$error_count" -gt 10 ]; then
                log_message WARN "High error count in $log_file: $error_count"
                send_alert "Log Alert" "High error count in $log_file: $error_count errors i
            fi
        fi
    done
}

# Generate system report
generate_report() {
    local report_file="/tmp/system_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "System Monitoring Report"
        echo "======"
        echo "Generated: $(date)"
        echo "Hostname: $(hostname)"
        echo

        echo "System Information:"
        echo "======"
        uname -a
        echo

        echo "Uptime:"
        echo "======"
        uptime
    }
}

```

```

    echo

    echo "CPU Information:"
    echo "======"
    lscpu | grep -E "Model name|CPU\(s\)|Thread|Core"
    echo

    echo "Memory Information:"
    echo "======"
    free -h
    echo

    echo "Disk Usage:"
    echo "======"
    df -h
    echo

    echo "Network Interfaces:"
    echo "======"
    ip addr show
    echo

    echo "Running Processes (Top 10 by CPU):"
    echo "======"
    ps aux --sort=-%cpu | head -11
    echo

    echo "Running Processes (Top 10 by Memory):"
    echo "======"
    ps aux --sort=-%mem | head -11

} > "$report_file"

echo "System report generated: $report_file"
log_message INFO "System report generated: $report_file"
}

# Main monitoring function
run_monitoring() {
    log_message INFO "Starting system monitoring"

    echo "System Monitoring Report - $(date)"
    echo "======"
    echo

    check_cpu
    echo

```

```

    check_memory
    echo

    check_disk
    echo

    check_load
    echo

    check_services
    echo

    check_network
    echo

    check_logs
    echo

    log_message INFO "System monitoring completed"
}

# Show usage
show_usage() {
    cat << EOF
Usage: $0 [options]

Options:
    -c, --continuous      Run monitoring continuously
    -i, --interval N      Set monitoring interval in seconds (default: 300)
    -r, --report          Generate detailed system report
    -h, --help            Show this help message

Environment Variables:
    ALERT_CPU_THRESHOLD    CPU usage alert threshold (default: 80)
    ALERT_MEMORY_THRESHOLD Memory usage alert threshold (default: 85)
    ALERT_DISK_THRESHOLD   Disk usage alert threshold (default: 90)
    LOG_FILE               Log file path (default: /var/log/system_monitor.log)
    EMAIL_ALERT            Email address for alerts
    WEBHOOK_URL            Webhook URL for alerts

Examples:
    $0                      # Run monitoring once
    $0 -c                  # Run continuous monitoring
    $0 -c -i 60            # Run continuous monitoring every 60 seconds
    $0 -r                  # Generate detailed report
EOF

```

```

}

# Parse command line arguments
CONTINUOUS=false
INTERVAL=300

while [[ $# -gt 0 ]]; do
    case $1 in
        -c|--continuous)
            CONTINUOUS=true
            shift
            ;;
        -i|--interval)
            INTERVAL="$2"
            shift 2
            ;;
        -r|--report)
            generate_report
            exit 0
            ;;
        -h|--help)
            show_usage
            exit 0
            ;;
        *)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
    esac
done

# Main execution
if [ "$CONTINUOUS" = true ]; then
    log_message INFO "Starting continuous monitoring (interval: ${INTERVAL}s)"
    while true; do
        run_monitoring
        echo "Sleeping for $INTERVAL seconds..."
        sleep "$INTERVAL"
    done
else
    run_monitoring
fi

```

This workbook provides a comprehensive collection of Bash scripting examples ranging from basic to intermediate level. Each example includes practical, real-world scenarios that demonstrate different aspects of Bash programming. The examples cover:

1. **Basic Examples:** Variables, user input, calculations, file operations
2. **Intermediate Examples:** Functions, configuration management, database operations
3. **File Processing:** Log analysis, file organization, CSV processing
4. **System Administration:** Comprehensive system monitoring

These examples can be used as learning materials, reference implementations, or starting points for your own scripts.

Advanced Bash Examples

This section contains advanced Bash scripting examples that demonstrate complex concepts, patterns, and real-world applications.

Process Management and IPC

Process Pool Manager

```
#!/bin/bash
# process_pool.sh - Manage a pool of worker processes

MAX_WORKERS=${MAX_WORKERS:-4}
WORK_QUEUE="/tmp/work_queue.$$"
WORKER_PIDS=()
RUNNING=true

# Cleanup function
cleanup() {
    RUNNING=false
    echo "Shutting down workers..."

    for pid in "${WORKER_PIDS[@]}; do
        if kill -0 "$pid" 2>/dev/null; then
            kill "$pid"
        fi
    done

    rm -f "$WORK_QUEUE"
    exit 0
}

# Set up signal handlers
trap cleanup SIGINT SIGTERM

# Worker function
worker() {
    local worker_id="$1"
    echo "Worker $worker_id started (PID: $$)"
```

```

while $RUNNING; do
    if [ -p "$WORK_QUEUE" ]; then
        if read -t 1 task < "$WORK_QUEUE"; then
            echo "Worker $worker_id processing: $task"
            # Simulate work
            sleep $((RANDOM % 5 + 1))
            echo "Worker $worker_id completed: $task"
        fi
    fi
done

echo "Worker $worker_id shutting down"
}

# Create work queue (named pipe)
mkfifo "$WORK_QUEUE"

# Start worker processes
for ((i=1; i<=MAX_WORKERS; i++)); do
    worker "$i" &
    WORKER_PIDS+=($!)
done

echo "Started $MAX_WORKERS workers"
echo "Send tasks by writing to: $WORK_QUEUE"
echo "Press Ctrl+C to shutdown"

# Add some sample tasks
for task in "task1" "task2" "task3" "task4" "task5"; do
    echo "$task" > "$WORK_QUEUE" &
done

# Wait for workers
wait
```#
Distributed Task Runner
```bash
#!/bin/bash
# distributed_runner.sh - Run tasks across multiple servers

SERVERS_FILE="${SERVERS_FILE:-servers.txt}"
SSH_KEY="${SSH_KEY:- ~/.ssh/id_rsa}"
PARALLEL_JOBS="${PARALLEL_JOBS:-3}"
TIMEOUT="${TIMEOUT:-300}"

# Server list format: user@hostname:port
declare -a SERVERS

```

```

# Load servers from file
load_servers() {
    if [ ! -f "$SERVERS_FILE" ]; then
        echo "Error: Servers file '$SERVERS_FILE' not found"
        exit 1
    fi

    while IFS= read -r server; do
        [[ $server =~ ^[[:space:]]*# ]] && continue
        [[ -z $server ]] && continue
        SERVERS+=("$server")
    done < "$SERVERS_FILE"

    echo "Loaded ${#SERVERS[@]} servers"
}

# Execute command on remote server
execute_remote() {
    local server="$1"
    local command="$2"
    local output_file="$3"

    local user_host="${server%:*}"
    local port="${server##*:}"
    [ "$port" = "$server" ] && port="22"

    echo "Executing on $user_host: $command" >> "$output_file"

    timeout "$TIMEOUT" ssh -i "$SSH_KEY" -p "$port" -o ConnectTimeout=10 \
        -o StrictHostKeyChecking=no "$user_host" "$command" >> "$output_file" 2>&1

    local exit_code=$?
    echo "Exit code: $exit_code" >> "$output_file"
    return $exit_code
}

# Run command on all servers in parallel
run_distributed() {
    local command="$1"
    local output_dir="results_$(date +%Y%m%d_%H%M%S)"

    mkdir -p "$output_dir"
    echo "Running command on ${#SERVERS[@]} servers: $command"
    echo "Output directory: $output_dir"

    local pids=()

```

```

local server_index=0

for server in "${SERVERS[@]}"; do
    local output_file="$output_dir/server_${server_index}.log"

    execute_remote "$server" "$command" "$output_file" &
    pids+=($!)

    ((server_index++))

    # Limit parallel jobs
    if [ ${#pids[@]} -ge $PARALLEL_JOBS ]; then
        wait "${pids[0]}"
        pids=("${pids[@]:1}")
    fi
done

# Wait for remaining jobs
for pid in "${pids[@]}"; do
    wait "$pid"
done

echo "All tasks completed. Results in: $output_dir"

# Generate summary
generate_summary "$output_dir"
}

# Generate execution summary
generate_summary() {
    local output_dir="$1"
    local summary_file="$output_dir/summary.txt"

    {
        echo "Distributed Execution Summary"
        echo "======"
        echo "Executed at: $(date)"
        echo "Total servers: ${#SERVERS[@]}"
        echo

        local success_count=0
        local failure_count=0

        for log_file in "$output_dir"/server_*.log; do
            local server_num=$(basename "$log_file" .log | cut -d_ -f2)
            local server="${SERVERS[$server_num]}"

```

```

        if grep -q "Exit code: 0" "$log_file"; then
            echo " $server - SUCCESS"
            ((success_count++))
        else
            echo " $server - FAILED"
            ((failure_count++))
        fi
    done

    echo
    echo "Results: $success_count successful, $failure_count failed"

} > "$summary_file"

cat "$summary_file"
}

# Main execution
if [ $# -eq 0 ]; then
    echo "Usage: $0 <command>"
    echo "Environment variables:"
    echo "  SERVERS_FILE    - File containing server list (default: servers.txt)"
    echo "  SSH_KEY         - SSH private key (default: ~/.ssh/id_rsa)"
    echo "  PARALLEL_JOBS   - Number of parallel jobs (default: 3)"
    echo "  TIMEOUT         - Command timeout in seconds (default: 300)"
    exit 1
fi

load_servers
run_distributed "$*"

```

Network Programming

HTTP Server in Bash

```

#!/bin/bash
# http_server.sh - Simple HTTP server in pure Bash

PORT="${PORT:-8080}"
DOCUMENT_ROOT="${DOCUMENT_ROOT:-./www}"
LOG_FILE="${LOG_FILE:-access.log}"

# MIME types
declare -A MIME_TYPES=(
    ["html"]="text/html"

```

```

        ["css"]="text/css"
        ["js"]="application/javascript"
        ["json"]="application/json"
        ["txt"]="text/plain"
        ["png"]="image/png"
        ["jpg"]="image/jpeg"
        ["gif"]="image/gif"
    )

# Log HTTP request
log_request() {
    local client_ip="$1"
    local method="$2"
    local path="$3"
    local status="$4"
    local size="$5"

    echo "$(date '+%d/%b/%Y:%H:%M:%S %z') $client_ip \"$method $path HTTP/1.1\" $status $size"
}

# Get MIME type from file extension
get_mime_type() {
    local file="$1"
    local extension="${file##*.}"
    echo "${MIME_TYPES[$extension]:-application/octet-stream}"
}

# Send HTTP response
send_response() {
    local status="$1"
    local content_type="$2"
    local content="$3"
    local content_length="${#content}"

    echo "HTTP/1.1 $status"
    echo "Content-Type: $content_type"
    echo "Content-Length: $content_length"
    echo "Connection: close"
    echo
    echo -n "$content"
}

# Handle HTTP request
handle_request() {
    local client_ip="$1"

    # Read request line

```

```

read -r method path protocol

# Read headers (until empty line)
while read -r header && [ -n "$header" ]; do
    header=$(echo "$header" | tr -d '\r')
    # Process headers if needed
done

# Remove query string from path
path="${path%/*\?*}"

# Security: prevent directory traversal
path="${path//..\./}"

# Default to index.html for directory requests
if [ "$path" == */ ]; then
    path="${path}index.html"
fi

local file_path="$DOCUMENT_ROOT$path"

if [ -f "$file_path" ]; then
    local content_type=$(get_mime_type "$file_path")
    local content=$(cat "$file_path")
    send_response "200 OK" "$content_type" "$content"
    log_request "$client_ip" "$method" "$path" "200" "${#content}"
else
    local error_content="<html><body><h1>404 Not Found</h1><p>The requested file was not
send_response "404 Not Found" "text/html" "$error_content"
log_request "$client_ip" "$method" "$path" "404" "${#error_content}"
fi
}

# Start server
start_server() {
    echo "Starting HTTP server on port $PORT"
    echo "Document root: $DOCUMENT_ROOT"
    echo "Log file: $LOG_FILE"

    # Create document root if it doesn't exist
    mkdir -p "$DOCUMENT_ROOT"

    # Create a simple index.html if it doesn't exist
    if [ ! -f "$DOCUMENT_ROOT/index.html" ]; then
        cat > "$DOCUMENT_ROOT/index.html" << EOF
<!DOCTYPE html>
<html>

```



```

<head>
  <title>Bash HTTP Server</title>
</head>
<body>
  <h1>Welcome to Bash HTTP Server</h1>
  <p>Server started at: $(date)</p>
  <p>This is a simple HTTP server written in Bash.</p>
</body>
</html>
EOF
    fi

    # Start listening
    while true; do
        # Use netcat to listen for connections
        if command -v nc >/dev/null 2>&1; then
            nc -l -p "$PORT" -e "$0" handle_connection
        elif command -v socat >/dev/null 2>&1; then
            socat TCP-LISTEN:$PORT,reuseaddr,fork EXEC:"$0 handle_connection"
        else
            echo "Error: netcat or socat required"
            exit 1
        fi
    done
}

# Handle individual connection
handle_connection() {
    local client_ip="${SOCAT_PEERADDR:-unknown}"
    handle_request "$client_ip"
}

# Main execution
case "${1:-start}" in
    start)
        start_server
        ;;
    handle_connection)
        handle_connection
        ;;
    *)
        echo "Usage: $0 [start]"
        exit 1
        ;;
esac

```

Network Scanner

```
#!/bin/bash
# network_scanner.sh - Network discovery and port scanning

NETWORK="${1:-192.168.1.0/24}"
TIMEOUT="${TIMEOUT:-1}"
THREADS="${THREADS:-50}"

# Common ports to scan
COMMON_PORTS=(22 23 25 53 80 110 143 443 993 995 3389 5432 3306)

# Parse CIDR notation
parse_network() {
    local network="$1"
    local ip="${network%/*}"
    local prefix="${network#*/}"

    if [ "$prefix" = "$network" ]; then
        # No CIDR notation, assume single host
        echo "$ip"
        return
    fi

    # Convert CIDR to IP range
    local IFS='.'
    local ip_parts=( $ip )
    local ip_int=$(( ${ip_parts[0]} << 24 | ${ip_parts[1]} << 16 | ${ip_parts[2]} << 8 | ${ip_parts[3]} ))

    local mask=$(( (0xFFFFFFFF << (32 - prefix)) ))
    local network_int=$(( ip_int & mask ))
    local broadcast_int=$(( network_int | (0xFFFFFFFF >> prefix) ))

    for ((i=network_int+1; i<broadcast_int; i++)); do
        printf "%d.%d.%d.%d\n" $((i >> 24 & 0xFF)) $((i >> 16 & 0xFF)) $((i >> 8 & 0xFF)) $(i & 0xFF)
    done
}

# Ping sweep to discover live hosts
ping_sweep() {
    local network="$1"
    local output_file="/tmp/live_hosts.$$"

    echo "Performing ping sweep on $network..."

    parse_network "$network" | while read -r ip; do
        {
```

```

        if ping -c 1 -W "$TIMEOUT" "$ip" >/dev/null 2>&1; then
            echo "$ip" >> "$output_file"
            echo "Host alive: $ip"
        fi
    } &

    # Limit concurrent processes
    (($ (jobs -r | wc -l) >= THREADS)) && wait
done

wait

if [ -f "$output_file" ]; then
    sort -V "$output_file"
    rm "$output_file"
fi
}

# Port scan a single host
port_scan() {
    local host="$1"
    local ports=("${@:2}")

    echo "Scanning ports on $host..."

    for port in "${ports[@]}; do
        {
            if timeout "$TIMEOUT" bash -c "echo >/dev/tcp/$host/$port" 2>/dev/null; then
                echo "$host:$port - OPEN"

                # Try to identify service
                local service=$(getent services "$port/tcp" 2>/dev/null | awk '{print $1}')
                [ -n "$service" ] && echo "  Service: $service"
            fi
        } &

        # Limit concurrent processes
        (($ (jobs -r | wc -l) >= THREADS)) && wait
    done

    wait
}

# Comprehensive network scan
full_scan() {
    local network="$1"
    local output_dir="scan_results_$(date +%Y%m%d_%H%M%S)"

```

```

mkdir -p "$output_dir"

echo "Starting comprehensive network scan..."
echo "Network: $network"
echo "Output directory: $output_dir"

# Discover live hosts
echo "Phase 1: Host discovery"
local live_hosts=()
while IFS= read -r host; do
    live_hosts+=("$host")
done < <(ping_sweep "$network")

echo "Found ${#live_hosts[@]} live hosts"
echo "${live_hosts[@]}" > "$output_dir/live_hosts.txt"

# Port scan each live host
echo "Phase 2: Port scanning"
for host in "${live_hosts[@]"; do
    echo "Scanning $host..."
    port_scan "$host" "${COMMON_PORTS[@]}" > "$output_dir/${host}_ports.txt"
done

# Generate summary report
{
    echo "Network Scan Report"
    echo "====="
    echo "Scan date: $(date)"
    echo "Network: $network"
    echo "Live hosts: ${#live_hosts[@]}"
    echo

    echo "Host Summary:"
    echo "====="
    for host in "${live_hosts[@]"; do
        local open_ports=$(grep -c "OPEN" "$output_dir/${host}_ports.txt" 2>/dev/null ||)
        echo "$host - $open_ports open ports"
    done

    echo
    echo "Open Ports by Service:"
    echo "====="
    for port in "${COMMON_PORTS[@]"; do
        local hosts_with_port=()
        for host in "${live_hosts[@]"; do
            if grep -q "$host:$port - OPEN" "$output_dir/${host}_ports.txt" 2>/dev/null;

```

```

        hosts_with_port+="$host")
    fi
done

    if [ ${#hosts_with_port[@]} -gt 0 ]; then
        local service=$(getent services "$port/tcp" 2>/dev/null | awk '{print $1}' |
        echo "Port $port ($service): ${hosts_with_port[*]}")
    fi
done

} > "$output_dir/summary.txt"

echo "Scan completed. Results in: $output_dir"
cat "$output_dir/summary.txt"
}

# Main execution
case "${2:-full}" in
    ping)
        ping_sweep "$NETWORK"
        ;;
    port)
        if [ $# -lt 2 ]; then
            echo "Usage: $0 <host> port [port...]"
            exit 1
        fi
        port_scan "$1" "${COMMON_PORTS[@]}"
        ;;
    full)
        full_scan "$NETWORK"
        ;;
    *)
        echo "Usage: $0 <network> [ping|port|full]"
        echo "Examples:"
        echo "  $0 192.168.1.0/24 ping      # Ping sweep only"
        echo "  $0 192.168.1.1 port        # Port scan single host"
        echo "  $0 192.168.1.0/24 full     # Full network scan"
        exit 1
        ;;
esac

```

Data Processing and Analysis

JSON Processor

```
#!/bin/bash
# json_processor.sh - JSON processing without external dependencies

# Simple JSON parser (basic implementation)
parse_json() {
    local json="$1"
    local key="$2"

    # Remove whitespace
    json=$(echo "$json" | tr -d ' \t\n\r')

    # Extract value for key
    local pattern="\$key\": \"([^\"]*)\""
    if [[ $json =~ $pattern ]]; then
        echo "${BASH_REMATCH[1]}"
    else
        # Try numeric value
        pattern="\$key\": ([0-9+)"
        if [[ $json =~ $pattern ]]; then
            echo "${BASH_REMATCH[1]}"
        fi
    fi
}

# Extract array elements
parse_json_array() {
    local json="$1"
    local array_key="$2"

    # Extract array content
    local pattern="\$array_key\": \[[^\]]*\]"
    if [[ $json =~ $pattern ]]; then
        local array_content="${BASH_REMATCH[1]}"

        # Split by comma and clean up
        IFS=',' read -ra elements <<< "$array_content"
        for element in "${elements[@]"; do
            element=$(echo "$element" | sed 's/^[[:space:]]*//;s/[[:space:]]*$//')
            echo "$element"
        done
    fi
}
```

```

# Create JSON object
create_json() {
    local -n data_ref=$1
    local json="{
    local first=true

    for key in "${!data_ref[@]}; do
        if [ "$first" = true ]; then
            first=false
        else
            json+=", "
        fi

        json+="\"$key\": \"${data_ref[$key]}\""
    done

    json+="}"
    echo "$json"
}

# Process JSON file
process_json_file() {
    local file="$1"
    local operation="$2"
    local key="$3"

    if [ ! -f "$file" ]; then
        echo "Error: File '$file' not found"
        return 1
    fi

    local json_content=$(cat "$file")

    case "$operation" in
        get)
            parse_json "$json_content" "$key"
            ;;
        array)
            parse_json_array "$json_content" "$key"
            ;;
        keys)
            # Extract all keys
            echo "$json_content" | grep -o '"[^"]*"[:space:]*:' | sed 's/"//g;s/[:space:]'
            ;;
        validate)
            # Basic JSON validation

```

```

        if echo "$json_content" | python3 -m json.tool >/dev/null 2>&1; then
            echo "Valid JSON"
        else
            echo "Invalid JSON"
        fi
        ;;
    pretty)
        # Pretty print JSON
        if command -v python3 >/dev/null 2>&1; then
            echo "$json_content" | python3 -m json.tool
        else
            echo "$json_content"
        fi
        ;;
    *)
        echo "Unknown operation: $operation"
        echo "Available operations: get, array, keys, validate, pretty"
        return 1
        ;;
esac
}

# Example usage and testing
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    # Create sample JSON file for testing
    cat > sample.json << 'EOF'
{
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "hobbies": ["reading", "swimming", "coding"],
    "active": true,
    "address": {
        "street": "123 Main St",
        "zip": "10001"
    }
}
EOF

    if [ $# -eq 0 ]; then
        echo "JSON Processor Demo"
        echo "===== "
        echo

        echo "Sample JSON content:"
        cat sample.json
        echo

```



```

    echo "Extracting 'name':"
    process_json_file sample.json get name
    echo

    echo "Extracting 'age':"
    process_json_file sample.json get age
    echo

    echo "Extracting 'hobbies' array:"
    process_json_file sample.json array hobbies
    echo

    echo "All keys:"
    process_json_file sample.json keys
    echo

    echo "Validating JSON:"
    process_json_file sample.json validate

else
    process_json_file "$@"
fi
fi

```

Data Aggregator

```

#!/bin/bash
# data_aggregator.sh - Aggregate and analyze data from multiple sources

declare -A metrics
declare -A counters
OUTPUT_FORMAT="${OUTPUT_FORMAT:-table}"

# Add metric
add_metric() {
    local key="$1"
    local value="$2"

    if [[ $value =~ ^[0-9]+(\.[0-9]+)?$ ]]; then
        metrics["$key"]="${metrics["$key"]:-0}"
        metrics["$key"]=$(echo "${metrics["$key"]} + $value" | bc -l)
        counters["$key"]=$(( ${counters["$key"]:-0} + 1 ))
    fi
}

```

```

# Process log file
process_log_file() {
    local file="$1"
    local format="$2"

    echo "Processing log file: $file (format: $format)"

    case "$format" in
        apache)
            # Apache access log format
            while IFS= read -r line; do
                # Extract response time if available
                if [[ $line =~ ([0-9]+)$ ]]; then
                    add_metric "response_time" "${BASH_REMATCH[1]}"
                fi

                # Extract status code
                if [[ $line =~ \"[^\"]*\"[[:space:]]+([0-9]{3}) ]]; then
                    add_metric "status_${BASH_REMATCH[1]}" 1
                fi

                # Extract bytes sent
                if [[ $line =~ [[:space:]]([0-9]+)[[:space:]]+\"[^\"]*\"[[:space:]]*$ ]]; then
                    add_metric "bytes_sent" "${BASH_REMATCH[1]}"
                fi
            done < "$file"
            ;;
        nginx)
            # Nginx access log format
            while IFS= read -r line; do
                # Similar processing for nginx logs
                if [[ $line =~ [[:space:]]([0-9]{3})[[:space:]] ]]; then
                    add_metric "status_${BASH_REMATCH[1]}" 1
                fi
            done < "$file"
            ;;
        csv)
            # CSV format - assume first line is header
            local headers=()
            local line_num=0

            while IFS=',' read -ra fields; do
                ((line_num++))

                if [ $line_num -eq 1 ]; then
                    headers=("${fields[@]}")
                else

```

```

        for i in "${!fields[@]}"; do
            local header="${headers[i]}"
            local value="${fields[i]}"

            if [[ $value =~ ^[0-9]+(\.[0-9]+)?$ ]]; then
                add_metric "$header" "$value"
            fi
        done
    fi
done < "$file"
;;
*)
    echo "Unknown format: $format"
    return 1
;;
esac
}

# Calculate statistics
calculate_stats() {
    declare -A averages
    declare -A totals

    for key in "${!metrics[@]}"; do
        totals["$key"]="${metrics["$key"]}"
        if [ "${counters["$key"]}" -gt 0 ]; then
            averages["$key"]=$(echo "scale=2; ${metrics["$key"]} / ${counters["$key"]}" | bc)
        fi
    done

    case "$OUTPUT_FORMAT" in
        table)
            printf "%-20s %15s %15s %10s\n" "Metric" "Total" "Average" "Count"
            printf "%-20s %15s %15s %10s\n" "-----" "-----" "-----" "-----"

            for key in $(printf '%s\n' "${!metrics[@]}" | sort); do
                printf "%-20s %15.2f %15.2f %10d\n" \
                    "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
            done
            ;;
        json)
            echo "{"
            local first=true
            for key in "${!metrics[@]}"; do
                if [ "$first" = true ]; then
                    first=false
                else

```

```

        echo -n ","
    fi
    printf '    "%s": {"total": %.2f, "average": %.2f, "count": %d}' \
        "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
done
echo
echo "}"
;;
csv)
    echo "metric,total,average,count"
    for key in "${!metrics[@]}"; do
        printf "%s,%.2f,%.2f,%d\n" \
            "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
    done
    ;;
esac
}

# Generate report
generate_report() {
    local output_file="$1"

    {
        echo "Data Aggregation Report"
        echo "======"
        echo "Generated: $(date)"
        echo "Files processed: ${#processed_files[@]}"
        echo

        calculate_stats

    } > "$output_file"

    echo "Report saved to: $output_file"
}

# Main processing function
main() {
    local files=("$@")
    local processed_files=()

    if [ ${#files[@]} -eq 0 ]; then
        echo "Usage: $0 <file1> [file2] ..."
        echo "Environment variables:"
        echo "    OUTPUT_FORMAT - Output format: table, json, csv (default: table)"
        exit 1
    fi

```

```

for file in "${files[@]"; do
    if [ ! -f "$file" ]; then
        echo "Warning: File '$file' not found, skipping"
        continue
    fi

    # Detect file format based on extension or content
    local format="apache" # default
    case "$file" in
        *.csv) format="csv" ;;
        *nginx*) format="nginx" ;;
        *apache*) format="apache" ;;
    esac

    process_log_file "$file" "$format"
    processed_files+=("$file")
done

if [ ${#processed_files[@]} -eq 0 ]; then
    echo "No files were processed"
    exit 1
fi

echo
echo "Aggregation Results:"
echo "======"
calculate_stats

# Generate detailed report
local report_file="aggregation_report_$(date +%Y%m%d_%H%M%S).txt"
generate_report "$report_file"
}

main "$@"

```

Security and Encryption

Password Manager

```

#!/bin/bash
# password_manager.sh - Simple encrypted password manager

VAULT_FILE="${VAULT_FILE:-$HOME/.password_vault.gpg}"
TEMP_FILE="/tmp/vault_temp.$$"

```

```

# Cleanup function
cleanup() {
    rm -f "$TEMP_FILE"
}
trap cleanup EXIT

# Check dependencies
check_dependencies() {
    if ! command -v gpg >/dev/null 2>&1; then
        echo "Error: GPG is required but not installed"
        exit 1
    fi
}

# Initialize vault
init_vault() {
    if [ -f "$VAULT_FILE" ]; then
        echo "Vault already exists at: $VAULT_FILE"
        return 1
    fi

    echo "Initializing new password vault..."
    echo "# Password Vault - Created $(date)" > "$TEMP_FILE"
    echo "# Format: service:username:password:notes" >> "$TEMP_FILE"

    encrypt_vault
    echo "Vault initialized at: $VAULT_FILE"
}

# Encrypt vault
encrypt_vault() {
    if ! gpg --symmetric --cipher-algo AES256 --output "$VAULT_FILE" "$TEMP_FILE"; then
        echo "Error: Failed to encrypt vault"
        exit 1
    fi
}

# Decrypt vault
decrypt_vault() {
    if [ ! -f "$VAULT_FILE" ]; then
        echo "Error: Vault file not found. Run 'init' first."
        exit 1
    fi

    if ! gpg --decrypt --output "$TEMP_FILE" "$VAULT_FILE" 2>/dev/null; then
        echo "Error: Failed to decrypt vault (wrong password?)"
    fi
}

```

```

        exit 1
    fi
}

# Add password entry
add_entry() {
    local service="$1"
    local username="$2"
    local password="$3"
    local notes="$4"

    if [ -z "$service" ] || [ -z "$username" ]; then
        echo "Error: Service and username are required"
        return 1
    fi

    decrypt_vault

    # Check if entry already exists
    if grep -q "^$service:$username:" "$TEMP_FILE"; then
        echo "Entry already exists. Use 'update' to modify."
        return 1
    fi

    # Generate password if not provided
    if [ -z "$password" ]; then
        password=$(generate_password)
        echo "Generated password: $password"
    fi

    echo "$service:$username:$password:$notes" >> "$TEMP_FILE"
    encrypt_vault

    echo "Entry added for $service ($username)"
}

# Generate secure password
generate_password() {
    local length="${1:-16}"
    tr -dc 'A-Za-z0-9!@#%&*~' < /dev/urandom | head -c "$length"
}

# Search entries
search_entries() {
    local query="$1"

    decrypt_vault

```

```

echo "Search results for: $query"
echo "======"

grep -i "$query" "$TEMP_FILE" | grep -v '^#' | while IFS=: read -r service username pa
    echo "Service: $service"
    echo "Username: $username"
    echo "Password: [hidden - use 'show' to reveal]"
    echo "Notes: $notes"
    echo "----"
done
}

# Show specific entry
show_entry() {
    local service="$1"
    local username="$2"

    decrypt_vault

    local entry=$(grep "^$service:$username:" "$TEMP_FILE")
    if [ -n "$entry" ]; then
        IFS=: read -r service username password notes <<< "$entry"
        echo "Service: $service"
        echo "Username: $username"
        echo "Password: $password"
        echo "Notes: $notes"
    else
        echo "Entry not found: $service ($username)"
    fi
}

# List all entries
list_entries() {
    decrypt_vault

    echo "Password Vault Entries"
    echo "======"

    grep -v '^#' "$TEMP_FILE" | while IFS=: read -r service username password notes; do
        echo "$service ($username)"
    done
}

# Update entry
update_entry() {
    local service="$1"

```



```

local username="$2"
local new_password="$3"
local new_notes="$4"

decrypt_vault

if ! grep -q "^$service:$username:" "$TEMP_FILE"; then
    echo "Entry not found: $service ($username)"
    return 1
fi

# Create updated file
local updated_file="/tmp/vault_updated.$$"

while IFS=: read -r s u p n; do
    if [ "$s" = "$service" ] && [ "$u" = "$username" ]; then
        echo "$s:$u:${new_password:-$p}:${new_notes:-$n}"
    else
        echo "$s:$u:$p:$n"
    fi
done < "$TEMP_FILE" > "$updated_file"

mv "$updated_file" "$TEMP_FILE"
encrypt_vault

echo "Entry updated: $service ($username)"
}

# Delete entry
delete_entry() {
    local service="$1"
    local username="$2"

    decrypt_vault

    if ! grep -q "^$service:$username:" "$TEMP_FILE"; then
        echo "Entry not found: $service ($username)"
        return 1
    fi

    # Create file without the entry
    grep -v "^$service:$username:" "$TEMP_FILE" > "/tmp/vault_filtered.$$"
    mv "/tmp/vault_filtered.$$" "$TEMP_FILE"
    encrypt_vault

    echo "Entry deleted: $service ($username)"
}

```

```

# Show usage
show_usage() {
    cat << EOF
Password Manager - Encrypted password storage

Usage: $0 <command> [arguments]

Commands:
    init                Initialize new vault
    add <service> <username> [password] [notes]  Add new entry
    show <service> <username>                Show specific entry
    search <query>                          Search entries
    list                                List all entries
    update <service> <username> [password] [notes]  Update entry
    delete <service> <username>                Delete entry
    generate [length]                        Generate secure password

Environment Variables:
    VAULT_FILE                Path to vault file (default: ~/.password_vault.gpg)

Examples:
    $0 init
    $0 add gmail john.doe@gmail.com
    $0 show gmail john.doe@gmail.com
    $0 search gmail
    $0 generate 20
EOF
}

# Main execution
check_dependencies

case "${1:-help}" in
    init)
        init_vault
        ;;
    add)
        add_entry "$2" "$3" "$4" "$5"
        ;;
    show)
        show_entry "$2" "$3"
        ;;
    search)
        search_entries "$2"
        ;;
    list)

```

```

        list_entries
        ;;
    update)
        update_entry "$2" "$3" "$4" "$5"
        ;;
    delete)
        delete_entry "$2" "$3"
        ;;
    generate)
        generate_password "$2"
        ;;
    help|--help|-h)
        show_usage
        ;;
    *)
        echo "Unknown command: $1"
        show_usage
        exit 1
        ;;
esac

```

These advanced examples demonstrate sophisticated Bash scripting techniques including process management, network programming, data processing, and security implementations. Each example showcases real-world applications and advanced programming patterns that can be adapted for various use cases.

Automation Examples

This section contains practical automation scripts for common system administration and development tasks.

System Administration Automation

Automated Backup System

```
#!/bin/bash
# automated_backup.sh - Comprehensive backup automation system

# Configuration
BACKUP_CONFIG="${BACKUP_CONFIG:-/etc/backup.conf}"
BACKUP_ROOT="${BACKUP_ROOT:-/backup}"
LOG_FILE="${LOG_FILE:-/var/log/backup.log}"
RETENTION_DAYS="${RETENTION_DAYS:-30}"
COMPRESSION="${COMPRESSION:-gzip}"
ENCRYPTION="${ENCRYPTION:-false}"
GPG_RECIPIENT="${GPG_RECIPIENT:-}"

# Default backup sources
declare -a BACKUP_SOURCES=(
    "/home"
    "/etc"
    "/var/www"
    "/opt"
)

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Load configuration
load_config() {
```

```

if [ -f "$BACKUP_CONFIG" ]; then
    source "$BACKUP_CONFIG"
    log INFO "Configuration loaded from $BACKUP_CONFIG"
else
    log WARN "Configuration file not found, using defaults"
fi
}

# Create backup directory structure
setup_backup_dirs() {
    local timestamp=$(date +%Y%m%d_%H%M%S)
    BACKUP_DIR="$BACKUP_ROOT/$timestamp"

    mkdir -p "$BACKUP_DIR"
    log INFO "Created backup directory: $BACKUP_DIR"
}

# Backup database
backup_database() {
    local db_type="$1"
    local db_name="$2"
    local db_user="$3"
    local db_host="${4:-localhost}"

    log INFO "Starting database backup: $db_type/$db_name"

    case "$db_type" in
        mysql)
            if command -v mysqldump >/dev/null 2>&1; then
                mysqldump -h "$db_host" -u "$db_user" -p"$DB_PASSWORD" "$db_name" > "$BACKUP_DIR/$db_name.sql"
                log INFO "MySQL database backup completed: $db_name"
            else
                log ERROR "mysqldump not found"
                return 1
            fi
            ;;
        postgresql)
            if command -v pg_dump >/dev/null 2>&1; then
                PGPASSWORD="$DB_PASSWORD" pg_dump -h "$db_host" -U "$db_user" "$db_name" > "$BACKUP_DIR/$db_name.sql"
                log INFO "PostgreSQL database backup completed: $db_name"
            else
                log ERROR "pg_dump not found"
                return 1
            fi
            ;;
    esac
}
*)

```

```

        log ERROR "Unsupported database type: $db_type"
        return 1
    ;;
esac
}

# Backup files and directories
backup_files() {
    local source="$1"
    local exclude_file="/tmp/backup_exclude.$$"

    # Create exclude file
    cat > "$exclude_file" << EOF
*.tmp
*.log
*.cache
*~
.DS_Store
Thumbs.db
EOF

    if [ -d "$source" ]; then
        local backup_name=$(basename "$source")
        local archive_file="$BACKUP_DIR/${backup_name}.tar"

        log INFO "Backing up directory: $source"

        if tar --exclude-from="$exclude_file" -cf "$archive_file" "$source"; then
            # Compress if requested
            if [ "$COMPRESSION" = "gzip" ]; then
                gzip "$archive_file"
                archive_file="${archive_file}.gz"
            elif [ "$COMPRESSION" = "bzip2" ]; then
                bzip2 "$archive_file"
                archive_file="${archive_file}.bz2"
            fi

            # Encrypt if requested
            if [ "$ENCRYPTION" = "true" ] && [ -n "$GPG_RECIPIENT" ]; then
                gpg --trust-model always --encrypt -r "$GPG_RECIPIENT" "$archive_file"
                rm "$archive_file"
                archive_file="${archive_file}.gpg"
            fi

            log INFO "Backup completed: $archive_file"
        else

```

```

        log ERROR "Backup failed for: $source"
    fi
else
    log WARN "Source not found: $source"
fi

rm -f "$exclude_file"
}

# Verify backup integrity
verify_backup() {
    local backup_file="$1"

    log INFO "Verifying backup: $backup_file"

    case "$backup_file" in
        *.tar.gz)
            if gzip -t "$backup_file" && tar -tzf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.tar.bz2)
            if bzip2 -t "$backup_file" && tar -tjf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.tar)
            if tar -tf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.gpg)
            if gpg --list-packets "$backup_file" >/dev/null 2>&1; then
                log INFO "Encrypted backup verification successful: $backup_file"
                return 0
            fi
            ;;
    esac

    log ERROR "Backup verification failed: $backup_file"
    return 1
}

# Clean old backups

```

```

cleanup_old_backups() {
    log INFO "Cleaning up backups older than $RETENTION_DAYS days"

    local deleted_count=0
    find "$BACKUP_ROOT" -maxdepth 1 -type d -name "20*" -mtime +$RETENTION_DAYS | while read
    do
        log INFO "Removing old backup: $old_backup"
        rm -rf "$old_backup"
        ((deleted_count++))
    done

    log INFO "Cleanup completed, removed $deleted_count old backups"
}

# Send notification
send_notification() {
    local status="$1"
    local summary="$2"

    if [ -n "$NOTIFICATION_EMAIL" ] && command -v mail >/dev/null 2>&1; then
    {
        echo "Backup Status: $status"
        echo "Time: $(date)"
        echo "Summary: $summary"
        echo
        echo "Recent log entries:"
        tail -20 "$LOG_FILE"
    } | mail -s "Backup Report - $status" "$NOTIFICATION_EMAIL"

    log INFO "Notification sent to $NOTIFICATION_EMAIL"
fi

    if [ -n "$SLACK_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
        local color="good"
        [ "$status" != "SUCCESS" ] && color="danger"

        curl -X POST -H 'Content-type: application/json' \
            --data '{"attachments":[{"color":"'${color}',"title":"'Backup Report',"text":"'
            "$SLACK_WEBHOOK" >/dev/null 2>&1

        log INFO "Slack notification sent"
    fi
}

# Generate backup report
generate_report() {
    local report_file="$BACKUP_DIR/backup_report.txt"
    local total_size=0

```



```

local file_count=0

{
    echo "Backup Report"
    echo "======"
    echo "Date: $(date)"
    echo "Backup Directory: $BACKUP_DIR"
    echo

    echo "Backed up files:"
    echo "======"

    for file in "$BACKUP_DIR"/*; do
        if [ -f "$file" ] && [[ "$(basename "$file")" != "backup_report.txt" ]]; then
            local size=$(stat -c%s "$file" 2>/dev/null || stat -f%z "$file" 2>/dev/null)
            local human_size=$(numfmt --to=iec-i --suffix=B "$size")

            echo "$(basename "$file"): $human_size"
            total_size=$((total_size + size))
            ((file_count++))
        fi
    done

    echo
    echo "Summary:"
    echo "======"
    echo "Total files: $file_count"
    echo "Total size: $(numfmt --to=iec-i --suffix=B $total_size)"
    echo "Compression: $COMPRESSION"
    echo "Encryption: $ENCRYPTION"

} > "$report_file"

log INFO "Backup report generated: $report_file"
}

# Main backup process
main() {
    log INFO "Starting automated backup process"

    load_config
    setup_backup_dirs

    local backup_success=true
    local backed_up_items=0

    # Backup files and directories

```

```

for source in "${BACKUP_SOURCES[@]}; do
    if backup_files "$source"; then
        ((backed_up_items++))
    else
        backup_success=false
    fi
done

# Backup databases if configured
if [ -n "$DATABASES" ]; then
    IFS=' ' read -ra DB_LIST <<< "$DATABASES"
    for db_config in "${DB_LIST[@]}; do
        IFS=':' read -ra DB_PARTS <<< "$db_config"
        if [ ${#DB_PARTS[@]} -ge 3 ]; then
            if backup_database "${DB_PARTS[0]}" "${DB_PARTS[1]}" "${DB_PARTS[2]}" "${DB_
                ((backed_up_items++))
            else
                backup_success=false
            fi
        fi
    done
fi

# Verify backups
for backup_file in "$BACKUP_DIR"/*; do
    if [ -f "$backup_file" ] && [[ "$(basename "$backup_file")" != "backup_report.txt" ]
        if ! verify_backup "$backup_file"; then
            backup_success=false
        fi
    fi
done

# Generate report
generate_report

# Cleanup old backups
cleanup_old_backups

# Send notification
if [ "$backup_success" = true ]; then
    send_notification "SUCCESS" "Backup completed successfully ($backed_up_items items)"
    log INFO "Backup process completed successfully"
else
    send_notification "FAILED" "Backup completed with errors"
    log ERROR "Backup process completed with errors"
    exit 1
fi

```

```

}

# Handle command line arguments
case "${1:-backup}" in
    backup)
        main
        ;;
    config)
        cat > "$BACKUP_CONFIG" << EOF
# Backup Configuration File
# Generated on $(date)

# Backup sources (space-separated)
BACKUP_SOURCES=(
    "/home"
    "/etc"
    "/var/www"
    "/opt"
)

# Database backups (format: type:name:user:host)
# DATABASES="mysql:myapp:dbuser:localhost,postgresql:webapp:pguser:localhost"

# Backup settings
RETENTION_DAYS=30
COMPRESSION="gzip" # gzip, bzip2, or none
ENCRYPTION="false"
GPG_RECIPIENT=""

# Notifications
NOTIFICATION_EMAIL=""
SLACK_WEBHOOK=""

# Database credentials
DB_PASSWORD=""
EOF

    echo "Configuration template created: $BACKUP_CONFIG"
    ;;
    restore)
        echo "Restore functionality not implemented yet"
        ;;
    *)
        echo "Usage: $0 {backup|config|restore}"
        echo "Environment variables:"
        echo "  BACKUP_CONFIG    - Configuration file path"
        echo "  BACKUP_ROOT      - Root backup directory"

```

```

        echo "    LOG_FILE          - Log file path"
        exit 1
    ;;
esac

```

System Health Monitor

```

#!/bin/bash
# health_monitor.sh - Comprehensive system health monitoring

MONITOR_CONFIG="${MONITOR_CONFIG:-/etc/health_monitor.conf}"
LOG_FILE="${LOG_FILE:-/var/log/health_monitor.log}"
ALERT_LOG="${ALERT_LOG:-/var/log/health_alerts.log}"
CHECK_INTERVAL="${CHECK_INTERVAL:-300}" # 5 minutes

# Thresholds
CPU_THRESHOLD="${CPU_THRESHOLD:-80}"
MEMORY_THRESHOLD="${MEMORY_THRESHOLD:-85}"
DISK_THRESHOLD="${DISK_THRESHOLD:-90}"
LOAD_THRESHOLD="${LOAD_THRESHOLD:-2.0}"
INODE_THRESHOLD="${INODE_THRESHOLD:-90}"

# Alert settings
ALERT_EMAIL="${ALERT_EMAIL:-}"
ALERT_WEBHOOK="${ALERT_WEBHOOK:-}"
ALERT_COOLDOWN="${ALERT_COOLDOWN:-3600}" # 1 hour

# Services to monitor
CRITICAL_SERVICES=("ssh" "nginx" "apache2" "mysql" "postgresql")

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Alert function
alert() {
    local severity="$1"
    local message="$2"
    local alert_key="$3"

    # Check cooldown

```

```

local last_alert_file="/tmp/alert_${alert_key}.last"
if [ -f "$last_alert_file" ]; then
    local last_alert=$(cat "$last_alert_file")
    local current_time=$(date +%s)
    local time_diff=$((current_time - last_alert))

    if [ $time_diff -lt $ALERT_COOLDOWN ]; then
        log DEBUG "Alert suppressed due to cooldown: $alert_key"
        return
    fi
fi

# Log alert
echo "$(date '+%Y-%m-%d %H:%M:%S') [$severity] $message" >> "$ALERT_LOG"
log WARN "ALERT [$severity]: $message"

# Send notifications
send_alert_notification "$severity" "$message"

# Update cooldown
date +%s > "$last_alert_file"
}

# Send alert notifications
send_alert_notification() {
    local severity="$1"
    local message="$2"

# Email notification
if [ -n "$ALERT_EMAIL" ] && command -v mail >/dev/null 2>&1; then
{
    echo "System Health Alert"
    echo "====="
    echo "Severity: $severity"
    echo "Time: $(date)"
    echo "Host: $(hostname)"
    echo
    echo "Message: $message"
    echo
    echo "System Information:"
    echo "====="
    uname -a
    echo
    echo "Load Average:"
    uptime
    echo

```

```

        echo "Memory Usage:"
        free -h
        echo
        echo "Disk Usage:"
        df -h
    } | mail -s "[${severity}] System Health Alert - ${hostname}" "$ALERT_EMAIL"
fi

# Webhook notification
if [ -n "$ALERT_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
    local color="warning"
    [ "${severity}" = "CRITICAL" ] && color="danger"

    curl -X POST -H 'Content-type: application/json' \
        --data '{"attachments":[{"color":"${color}","title":"System Health Alert\
"$ALERT_WEBHOOK" >/dev/null 2>&1
fi
}

# Check CPU usage
check_cpu() {
    local cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)
    cpu_usage=${cpu_usage%. *}

    log INFO "CPU usage: ${cpu_usage}%"

    if [ "${cpu_usage}" -gt "$CPU_THRESHOLD" ]; then
        alert "WARNING" "High CPU usage: ${cpu_usage}% (threshold: ${CPU_THRESHOLD}%)" "cpu_"

        # Get top CPU processes
        local top_processes=$(ps aux --sort=-%cpu | head -6 | tail -5 | awk '{print $11 " ("')
        log INFO "Top CPU processes: $top_processes"
    fi

    echo "cpu_usage:$cpu_usage"
}

# Check memory usage
check_memory() {
    local memory_info=$(free | grep Mem)
    local total=$(echo $memory_info | awk '{print $2}')
    local used=$(echo $memory_info | awk '{print $3}')
    local available=$(echo $memory_info | awk '{print $7}')
    local memory_usage=$((used * 100 / total))

    log INFO "Memory usage: ${memory_usage}% (${used}/${total})"
}

```

```

if [ "$memory_usage" -gt "$MEMORY_THRESHOLD" ]; then
    alert "WARNING" "High memory usage: ${memory_usage}% (threshold: ${MEMORY_THRESHOLD})"

    # Get top memory processes
    local top_processes=$(ps aux --sort=-%mem | head -6 | tail -5 | awk '{print $11 " ("')
    log INFO "Top memory processes: $top_processes"
fi

echo "memory_usage:$memory_usage"
echo "memory_available:$available"
}

# Check disk usage
check_disk() {
    local alerts_sent=false

    df -h | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        local usage=$(echo $output | awk '{print $5}' | cut -d '%' -f1)
        local partition=$(echo $output | awk '{print $1}')
        local mount_point=$(echo $output | awk '{print $6}')
        local size=$(echo $output | awk '{print $2}')
        local available=$(echo $output | awk '{print $4}')

        log INFO "Disk usage $mount_point: ${usage}% ($available available)"

        if [ "$usage" -gt "$DISK_THRESHOLD" ]; then
            alert "WARNING" "High disk usage on $mount_point: ${usage}% (threshold: ${DISK_THRESHOLD})"
            alerts_sent=true
        fi

        echo "disk_usage_${mount_point//\//\_}:$usage"
    done

    # Check inode usage
    df -i | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        local inode_usage=$(echo $output | awk '{print $5}' | cut -d '%' -f1)
        local mount_point=$(echo $output | awk '{print $6}')

        if [ "$inode_usage" -gt "$INODE_THRESHOLD" ]; then
            alert "WARNING" "High inode usage on $mount_point: ${inode_usage}% (threshold: ${INODE_THRESHOLD})"
        fi

        echo "inode_usage_${mount_point//\//\_}:$inode_usage"
    done
}

```

```

# Check system load
check_load() {
    local load_1min=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}' | tr -d ' ')
    local cpu_cores=$(nproc)
    local load_per_core=$(echo "scale=2; $load_1min / $cpu_cores" | bc -l)

    log INFO "Load average: $load_1min (${load_per_core} per core)"

    if (( $(echo "$load_per_core > $LOAD_THRESHOLD" | bc -l) )); then
        alert "WARNING" "High system load: $load_1min (${load_per_core} per core, threshold: $LOAD_THRESHOLD)"
    fi

    echo "load_1min:$load_1min"
    echo "load_per_core:$load_per_core"
}

# Check critical services
check_services() {
    local failed_services=()

    for service in "${CRITICAL_SERVICES[@]}; do
        if systemctl is-active --quiet "$service" 2>/dev/null; then
            log INFO "Service $service: running"
            echo "service_${service}:1"
        elif systemctl list-unit-files | grep -q "^$service.service"; then
            log WARN "Service $service: stopped"
            failed_services+=("$service")
            echo "service_${service}:0"
        else
            log DEBUG "Service $service: not installed"
            echo "service_${service}:-1"
        fi
    done

    if [ ${#failed_services[@]} -gt 0 ]; then
        alert "CRITICAL" "Critical services not running: ${failed_services[*]}" "services_down"
    fi
}

# Check network connectivity
check_network() {
    local test_hosts=("8.8.8.8" "1.1.1.1")
    local failed_hosts=()

    for host in "${test_hosts[@]}; do
        if ping -c 1 -W 5 "$host" >/dev/null 2>&1; then
            log INFO "Network connectivity to $host: OK"
        else
            failed_hosts+=("$host")
        fi
    done

    if [ ${#failed_hosts[@]} -gt 0 ]; then
        alert "CRITICAL" "Network connectivity failed to: ${failed_hosts[*]}" "network_down"
    fi
}

```



```

        echo "network_${host//./_}:1"
    else
        log WARN "Network connectivity to $host: FAILED"
        failed_hosts+=("$host")
        echo "network_${host//./_}:0"
    fi
done

if [ ${#failed_hosts[@]} -gt 0 ]; then
    alert "CRITICAL" "Network connectivity issues: ${failed_hosts[*]}" "network_down"
fi
}

# Check SSL certificates
check_ssl_certificates() {
    local cert_paths=("/etc/ssl/certs" "/etc/letsencrypt/live")
    local expiring_certs=()

    for cert_path in "${cert_paths[@]"; do
        if [ -d "$cert_path" ]; then
            find "$cert_path" -name "*.crt" -o -name "cert.pem" | while read -r cert_file; do
                if [ -f "$cert_file" ]; then
                    local expiry_date=$(openssl x509 -in "$cert_file" -noout -enddate 2>/dev/
                    if [ -n "$expiry_date" ]; then
                        local expiry_timestamp=$(date -d "$expiry_date" +%s 2>/dev/null)
                        local current_timestamp=$(date +%s)
                        local days_until_expiry=$(( (expiry_timestamp - current_timestamp) /

                        log INFO "SSL certificate $cert_file expires in $days_until_expiry d

                        if [ "$days_until_expiry" -lt 30 ]; then
                            expiring_certs+=("${(basename "$cert_file")}:$days_until_expiry")
                        fi

                        echo "ssl_cert_${(basename "$cert_file" .crt | tr '.' '_')}_days:$days

                    fi
                fi
            done
        fi
    done

    if [ ${#expiring_certs[@]} -gt 0 ]; then
        alert "WARNING" "SSL certificates expiring soon: ${expiring_certs[*]}" "ssl_expiring
    fi
}

```

```

# Check log files for errors
check_log_errors() {
    local log_files=("/var/log/syslog" "/var/log/auth.log" "/var/log/kern.log")
    local error_threshold=50
    local time_window="1 hour ago"

    for log_file in "${log_files[@]"; do
        if [ -f "$log_file" ]; then
            local error_count=$(find "$log_file" -newermt "$time_window" -exec grep -i "error" {} \; | wc -l)

            log INFO "Errors in $(basename "$log_file"): $error_count (last hour)"

            if [ "$error_count" -gt "$error_threshold" ]; then
                alert "WARNING" "High error count in $log_file: $error_count errors in last hour"
            fi

            echo "log_errors_$(basename "$log_file"):$error_count"
        fi
    done
}

# Generate metrics for monitoring systems
generate_metrics() {
    local metrics_file="/tmp/health_metrics.$$"

    {
        check_cpu
        check_memory
        check_disk
        check_load
        check_services
        check_network
        check_ssl_certificates
        check_log_errors
    } > "$metrics_file"

    # Output in Prometheus format if requested
    if [ "$OUTPUT_FORMAT" = "prometheus" ]; then
        while IFS=: read -r metric value; do
            echo "system_${metric} ${value}"
        done < "$metrics_file"
    else
        cat "$metrics_file"
    fi

    rm -f "$metrics_file"
}

```

```

# Main monitoring loop
monitor_loop() {
    log INFO "Starting health monitoring loop (interval: ${CHECK_INTERVAL}s)"

    while true; do
        log INFO "Running health checks..."

        generate_metrics > "/tmp/current_metrics.txt"

        log INFO "Health check completed"

        sleep "$CHECK_INTERVAL"
    done
}

# One-time health check
health_check() {
    log INFO "Running one-time health check"
    generate_metrics
}

# Show system status dashboard
show_dashboard() {
    clear
    echo "System Health Dashboard - $(hostname)"
    echo "====="
    echo "Last updated: $(date)"
    echo

    # System info
    echo "System Information:"
    echo "====="
    echo "Uptime: $(uptime -p)"
    echo "Kernel: $(uname -r)"
    echo "Architecture: $(uname -m)"
    echo

    # Current metrics
    echo "Current Metrics:"
    echo "====="
    generate_metrics | while IFS=: read -r metric value; do
        printf "%-30s: %s\n" "$metric" "$value"
    done

    echo
    echo "Recent Alerts:"

```

```

        echo "======"
        if [ -f "$ALERT_LOG" ]; then
            tail -5 "$ALERT_LOG"
        else
            echo "No recent alerts"
        fi
    }

# Load configuration
load_config() {
    if [ -f "$MONITOR_CONFIG" ]; then
        source "$MONITOR_CONFIG"
        log INFO "Configuration loaded from $MONITOR_CONFIG"
    fi
}

# Main execution
load_config

case "${1:-check}" in
    check)
        health_check
        ;;
    monitor)
        monitor_loop
        ;;
    dashboard)
        show_dashboard
        ;;
    metrics)
        OUTPUT_FORMAT="prometheus" generate_metrics
        ;;
    config)
        cat > "$MONITOR_CONFIG" << EOF
# Health Monitor Configuration
# Generated on $(date)

# Check intervals (seconds)
CHECK_INTERVAL=300

# Thresholds
CPU_THRESHOLD=80
MEMORY_THRESHOLD=85
DISK_THRESHOLD=90
LOAD_THRESHOLD=2.0
INODE_THRESHOLD=90

```

```

# Alert settings
ALERT_EMAIL=""
ALERT_WEBHOOK=""
ALERT_COOLDOWN=3600

# Critical services to monitor
CRITICAL_SERVICES=("ssh" "nginx" "apache2" "mysql" "postgresql")

# Log files
LOG_FILE="/var/log/health_monitor.log"
ALERT_LOG="/var/log/health_alerts.log"
EOF

    echo "Configuration template created: $MONITOR_CONFIG"
    ;;
*)
    echo "Usage: $0 {check|monitor|dashboard|metrics|config}"
    echo "Commands:"
    echo "  check      - Run one-time health check"
    echo "  monitor    - Start continuous monitoring"
    echo "  dashboard  - Show system status dashboard"
    echo "  metrics    - Output metrics in Prometheus format"
    echo "  config     - Generate configuration template"
    exit 1
    ;;
esac
```##
Development Automation

CI/CD Pipeline Script
```bash
#!/bin/bash
# cisd_pipeline.sh - Continuous Integration/Continuous Deployment pipeline

PROJECT_NAME="${PROJECT_NAME:-myapp}"
GIT_REPO="${GIT_REPO:-}"
BUILD_DIR="${BUILD_DIR:-./build}"
DEPLOY_DIR="${DEPLOY_DIR:-/var/www/$PROJECT_NAME}"
LOG_FILE="${LOG_FILE:-./pipeline.log}"

# Pipeline stages
STAGES=("checkout" "test" "build" "deploy" "notify")

# Configuration
SLACK_WEBHOOK="${SLACK_WEBHOOK:-}"
EMAIL_NOTIFICATIONS="${EMAIL_NOTIFICATIONS:-}"
DOCKER_REGISTRY="${DOCKER_REGISTRY:-}"
KUBERNETES_NAMESPACE="${KUBERNETES_NAMESPACE:-default}"

```

```

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
NC='\033[0m'

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    echo "[${timestamp}] [${level}] $message" | tee -a "$LOG_FILE"

    case "$level" in
        ERROR) echo -e "${RED}[${timestamp}] ERROR: $message${NC}" ;;
        WARN)  echo -e "${YELLOW}[${timestamp}] WARN: $message${NC}" ;;
        INFO)  echo -e "${GREEN}[${timestamp}] INFO: $message${NC}" ;;
        DEBUG) echo -e "${BLUE}[${timestamp}] DEBUG: $message${NC}" ;;
    esac
}

# Send notification
send_notification() {
    local status="$1"
    local stage="$2"
    local message="$3"

    # Slack notification
    if [ -n "$SLACK_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
        local color="good"
        [ "$status" != "SUCCESS" ] && color="danger"

        curl -X POST -H 'Content-type: application/json' \
            --data '{"attachments":[{"color":"${color}","title":"Pipeline $status","\
"$SLACK_WEBHOOK" >/dev/null 2>&1
    fi

    # Email notification
    if [ -n "$EMAIL_NOTIFICATIONS" ] && command -v mail >/dev/null 2>&1; then
        {
            echo "Pipeline Status: $status"
            echo "Project: $PROJECT_NAME"
            echo "Stage: $stage"
        }
    fi
}

```

```

        echo "Time: $(date)"
        echo
        echo "Message: $message"
        echo
        echo "Recent log entries:"
        tail -20 "$LOG_FILE"
    } | mail -s "Pipeline $status - $PROJECT_NAME" "$EMAIL_NOTIFICATIONS"
fi
}

# Checkout stage
stage_checkout() {
    log INFO "Starting checkout stage"

    if [ -z "$GIT_REPO" ]; then
        log ERROR "GIT_REPO not specified"
        return 1
    fi

    # Clean previous build
    rm -rf "$BUILD_DIR"
    mkdir -p "$BUILD_DIR"

    # Clone repository
    if git clone "$GIT_REPO" "$BUILD_DIR"; then
        log INFO "Repository cloned successfully"

        # Get commit information
        cd "$BUILD_DIR"
        local commit_hash=$(git rev-parse HEAD)
        local commit_message=$(git log -1 --pretty=%B)
        local author=$(git log -1 --pretty=%an)

        log INFO "Commit: $commit_hash"
        log INFO "Author: $author"
        log INFO "Message: $commit_message"

        return 0
    else
        log ERROR "Failed to clone repository"
        return 1
    fi
}

# Test stage
stage_test() {

```

```

log INFO "Starting test stage"

cd "$BUILD_DIR"

# Detect project type and run appropriate tests
if [ -f "package.json" ]; then
    # Node.js project
    log INFO "Detected Node.js project"

    if command -v npm >/dev/null 2>&1; then
        log INFO "Installing dependencies..."
        if npm install; then
            log INFO "Dependencies installed"
        else
            log ERROR "Failed to install dependencies"
            return 1
        fi

        log INFO "Running tests..."
        if npm test; then
            log INFO "Tests passed"
        else
            log ERROR "Tests failed"
            return 1
        fi
    else
        log ERROR "npm not found"
        return 1
    fi

elif [ -f "requirements.txt" ] || [ -f "setup.py" ]; then
    # Python project
    log INFO "Detected Python project"

    if command -v python3 >/dev/null 2>&1; then
        # Create virtual environment
        python3 -m venv venv
        source venv/bin/activate

        # Install dependencies
        if [ -f "requirements.txt" ]; then
            log INFO "Installing Python dependencies..."
            if pip install -r requirements.txt; then
                log INFO "Dependencies installed"
            else
                log ERROR "Failed to install dependencies"
                return 1
            fi
        fi
    fi

```



```

        fi
    fi

    # Run tests
    if [ -f "pytest.ini" ] || [ -d "tests" ]; then
        log INFO "Running Python tests..."
        if python -m pytest; then
            log INFO "Tests passed"
        else
            log ERROR "Tests failed"
            return 1
        fi
    fi
else
    log ERROR "python3 not found"
    return 1
fi

elif [ -f "Makefile" ]; then
    # Make-based project
    log INFO "Detected Make-based project"

    if make test; then
        log INFO "Tests passed"
    else
        log ERROR "Tests failed"
        return 1
    fi

else
    log WARN "No recognized test framework found, skipping tests"
fi

return 0
}

# Build stage
stage_build() {
    log INFO "Starting build stage"

    cd "$BUILD_DIR"

    # Detect project type and build
    if [ -f "Dockerfile" ]; then
        # Docker build
        log INFO "Building Docker image"
    fi
}

```

```

local image_tag="$PROJECT_NAME:${git rev-parse --short HEAD}"

if docker build -t "$image_tag" .; then
    log INFO "Docker image built: $image_tag"

    # Push to registry if configured
    if [ -n "$DOCKER_REGISTRY" ]; then
        local registry_image="$DOCKER_REGISTRY/$image_tag"

        if docker tag "$image_tag" "$registry_image"; then
            log INFO "Tagged image for registry: $registry_image"

            if docker push "$registry_image"; then
                log INFO "Image pushed to registry"
            else
                log ERROR "Failed to push image to registry"
                return 1
            fi
        fi
    fi
else
    log ERROR "Docker build failed"
    return 1
fi

elif [ -f "package.json" ]; then
    # Node.js build
    log INFO "Building Node.js application"

    if npm run build 2>/dev/null || npm run compile 2>/dev/null; then
        log INFO "Build completed"
    else
        log WARN "No build script found or build failed"
    fi

elif [ -f "Makefile" ]; then
    # Make build
    log INFO "Building with Make"

    if make; then
        log INFO "Build completed"
    else
        log ERROR "Build failed"
        return 1
    fi

else

```

```

    log INFO "No build process detected, copying files"

    # Simple file copy
    mkdir -p dist
    cp -r . dist/
    log INFO "Files copied to dist/"
fi

return 0
}

# Deploy stage
stage_deploy() {
    log INFO "Starting deploy stage"

    cd "$BUILD_DIR"

    # Backup current deployment
    if [ -d "$DEPLOY_DIR" ]; then
        local backup_dir="${DEPLOY_DIR}.backup.$(date +%Y%m%d_%H%M%S)"
        log INFO "Creating backup: $backup_dir"
        cp -r "$DEPLOY_DIR" "$backup_dir"
    fi

    # Deploy based on project type
    if [ -f "Dockerfile" ] && command -v kubectl >/dev/null 2>&1; then
        # Kubernetes deployment
        log INFO "Deploying to Kubernetes"

        local image_tag="$PROJECT_NAME:$(git rev-parse --short HEAD)"

        # Update deployment
        if kubectl set image deployment/$PROJECT_NAME container=$DOCKER_REGISTRY/$image_tag
            log INFO "Kubernetes deployment updated"

            # Wait for rollout
            if kubectl rollout status deployment/$PROJECT_NAME -n $KUBERNETES_NAMESPACE; then
                log INFO "Deployment rollout completed"
            else
                log ERROR "Deployment rollout failed"
                return 1
            fi
        else
            log ERROR "Failed to update Kubernetes deployment"
            return 1
        fi
    fi
}

```

```

else
    # File-based deployment
    log INFO "Deploying files to $DEPLOY_DIR"

    # Create deploy directory
    mkdir -p "$DEPLOY_DIR"

    # Copy files
    if [ -d "dist" ]; then
        cp -r dist/* "$DEPLOY_DIR/"
    else
        cp -r . "$DEPLOY_DIR/"
    fi

    # Set permissions
    chown -R www-data:www-data "$DEPLOY_DIR" 2>/dev/null || true
    chmod -R 755 "$DEPLOY_DIR"

    # Restart services if needed
    if systemctl is-active --quiet nginx; then
        systemctl reload nginx
        log INFO "Nginx reloaded"
    fi

    if systemctl is-active --quiet apache2; then
        systemctl reload apache2
        log INFO "Apache reloaded"
    fi

    log INFO "Deployment completed"
fi

return 0
}

# Notify stage
stage_notify() {
    log INFO "Starting notify stage"

    local commit_hash=$(cd "$BUILD_DIR" && git rev-parse --short HEAD)
    local commit_message=$(cd "$BUILD_DIR" && git log -1 --pretty=%B | head -1)

    send_notification "SUCCESS" "deploy" "Deployment completed successfully. Commit: $commit"

    return 0
}

```

```

# Run pipeline
run_pipeline() {
    local start_time=$(date +%s)
    log INFO "Starting CI/CD pipeline for $PROJECT_NAME"

    for stage in "${STAGES[@]}; do
        log INFO "Running stage: $stage"

        local stage_start=$(date +%s)

        if "stage_$stage"; then
            local stage_end=$(date +%s)
            local stage_duration=$((stage_end - stage_start))
            log INFO "Stage $stage completed in ${stage_duration}s"
        else
            local stage_end=$(date +%s)
            local stage_duration=$((stage_end - stage_start))
            log ERROR "Stage $stage failed after ${stage_duration}s"

            send_notification "FAILED" "$stage" "Pipeline failed at stage: $stage"
            exit 1
        fi
    done

    local end_time=$(date +%s)
    local total_duration=$((end_time - start_time))

    log INFO "Pipeline completed successfully in ${total_duration}s"
}

# Rollback function
rollback() {
    local backup_dir="$1"

    if [ -z "$backup_dir" ]; then
        # Find latest backup
        backup_dir=$(ls -td "${DEPLOY_DIR}.backup.* 2>/dev/null | head -1)
    fi

    if [ -d "$backup_dir" ]; then
        log INFO "Rolling back to: $backup_dir"

        rm -rf "$DEPLOY_DIR"
        cp -r "$backup_dir" "$DEPLOY_DIR"

        log INFO "Rollback completed"
    else

```

```

        log ERROR "Backup directory not found: $backup_dir"
        exit 1
    fi
}

# Show pipeline status
show_status() {
    echo "CI/CD Pipeline Status"
    echo "======"
    echo "Project: $PROJECT_NAME"
    echo "Build Directory: $BUILD_DIR"
    echo "Deploy Directory: $DEPLOY_DIR"
    echo

    if [ -f "$LOG_FILE" ]; then
        echo "Recent Activity:"
        echo "======"
        tail -20 "$LOG_FILE"
    fi

    echo
    echo "Available Backups:"
    echo "======"
    ls -la "${DEPLOY_DIR}.backup.* 2>/dev/null || echo "No backups found"
}

# Main execution
case "${1:-run}" in
    run)
        run_pipeline
        ;;
    checkout)
        stage_checkout
        ;;
    test)
        stage_test
        ;;
    build)
        stage_build
        ;;
    deploy)
        stage_deploy
        ;;
    rollback)
        rollback "$2"
        ;;
    status)

```

```

        show_status
        ;;
    *)
        echo "Usage: $0 {run|checkout|test|build|deploy|rollback|status}"
        echo "Environment variables:"
        echo "  PROJECT_NAME      - Project name"
        echo "  GIT_REPO           - Git repository URL"
        echo "  BUILD_DIR          - Build directory"
        echo "  DEPLOY_DIR         - Deployment directory"
        echo "  SLACK_WEBHOOK       - Slack webhook URL"
        echo "  EMAIL_NOTIFICATIONS - Email for notifications"
        echo "  DOCKER_REGISTRY    - Docker registry URL"
        echo "  KUBERNETES_NAMESPACE - Kubernetes namespace"
        exit 1
        ;;
esac

```

Environment Provisioning Script

```

#!/bin/bash
# provision_environment.sh - Automated environment provisioning

ENVIRONMENT="${1:-development}"
CONFIG_DIR="${CONFIG_DIR:-./config}"
ANSIBLE_PLAYBOOK="${ANSIBLE_PLAYBOOK:-}"
TERRAFORM_DIR="${TERRAFORM_DIR:-./terraform}"

# Environment configurations
declare -A ENV_CONFIGS=(
    ["development"]="dev"
    ["staging"]="stage"
    ["production"]="prod"
)

# Required tools
REQUIRED_TOOLS=("ansible" "terraform" "kubectl" "docker")

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*"
}

# Check prerequisites
check_prerequisites() {
    log "Checking prerequisites..."
}

```

```

local missing_tools=()

for tool in "${REQUIRED_TOOLS[@]"; do
    if ! command -v "$tool" >/dev/null 2>&1; then
        missing_tools+=("$tool")
    fi
done

if [ ${#missing_tools[@]} -gt 0 ]; then
    log "ERROR: Missing required tools: ${missing_tools[*]}"
    log "Please install the missing tools and try again."
    exit 1
fi

log "All prerequisites satisfied"
}

# Load environment configuration
load_environment_config() {
    local env="$1"
    local config_file="$CONFIG_DIR/${env}.conf"

    if [ -f "$config_file" ]; then
        source "$config_file"
        log "Loaded configuration for environment: $env"
    else
        log "WARNING: Configuration file not found: $config_file"
        log "Using default configuration"
    fi
}

# Provision infrastructure with Terraform
provision_infrastructure() {
    local env="$1"

    if [ ! -d "$TERRAFORM_DIR" ]; then
        log "Terraform directory not found: $TERRAFORM_DIR"
        return 1
    fi

    log "Provisioning infrastructure for environment: $env"

    cd "$TERRAFORM_DIR"

    # Initialize Terraform
    if terraform init; then

```



```

        log "Terraform initialized"
    else
        log "ERROR: Terraform initialization failed"
        return 1
    fi

    # Select workspace
    if terraform workspace select "$env" 2>/dev/null || terraform workspace new "$env"; then
        log "Terraform workspace: $env"
    else
        log "ERROR: Failed to select/create workspace: $env"
        return 1
    fi

    # Plan infrastructure changes
    log "Planning infrastructure changes..."
    if terraform plan -var-file="$CONFIG_DIR/${env}.tfvars" -out="$env.tfplan"; then
        log "Terraform plan completed"
    else
        log "ERROR: Terraform planning failed"
        return 1
    fi

    # Apply infrastructure changes
    log "Applying infrastructure changes..."
    if terraform apply "$env.tfplan"; then
        log "Infrastructure provisioned successfully"

        # Save outputs
        terraform output -json > "$CONFIG_DIR/${env}_outputs.json"
        log "Terraform outputs saved"
    else
        log "ERROR: Terraform apply failed"
        return 1
    fi

    cd - >/dev/null
}

# Configure servers with Ansible
configure_servers() {
    local env="$1"
    local playbook="${ANSIBLE_PLAYBOOK:-site.yml}"

    if [ ! -f "$playbook" ]; then
        log "Ansible playbook not found: $playbook"
        return 1
    fi

```

```

fi

log "Configuring servers for environment: $env"

# Run Ansible playbook
if ansible-playbook -i "inventory/${env}" "$playbook" --extra-vars "environment=$env"; then
    log "Server configuration completed"
else
    log "ERROR: Ansible playbook execution failed"
    return 1
fi
}

# Deploy applications
deploy_applications() {
    local env="$1"

    log "Deploying applications for environment: $env"

    # Deploy to Kubernetes if configured
    if [ -n "$KUBERNETES_NAMESPACE" ] && command -v kubectl >/dev/null 2>&1; then
        log "Deploying to Kubernetes namespace: $KUBERNETES_NAMESPACE"

        # Apply Kubernetes manifests
        if [ -d "k8s/$env" ]; then
            kubectl apply -f "k8s/$env/" -n "$KUBERNETES_NAMESPACE"
            log "Kubernetes manifests applied"
        fi

        # Wait for deployments to be ready
        kubectl wait --for=condition=available --timeout=300s deployment --all -n "$KUBERNETES_NAMESPACE"
        log "All deployments are ready"
    fi

    # Deploy with Docker Compose if configured
    if [ -f "docker-compose.${env}.yaml" ]; then
        log "Deploying with Docker Compose"

        docker-compose -f "docker-compose.${env}.yaml" up -d
        log "Docker Compose deployment completed"
    fi
}

# Setup monitoring and logging
setup_monitoring() {
    local env="$1"

```

```

log "Setting up monitoring for environment: $env"

# Deploy monitoring stack
if [ -d "monitoring/$env" ]; then
    kubectl apply -f "monitoring/$env/" -n "monitoring-$env" 2>/dev/null || true
    log "Monitoring stack deployed"
fi

# Configure log aggregation
if [ -f "logging/fluentd-${env}.yaml" ]; then
    kubectl apply -f "logging/fluentd-${env}.yaml" -n "logging-$env" 2>/dev/null || true
    log "Log aggregation configured"
fi
}

# Run health checks
run_health_checks() {
    local env="$1"

    log "Running health checks for environment: $env"

    # Load health check endpoints from configuration
    local health_endpoints=()
    if [ -f "$CONFIG_DIR/${env}_health_checks.txt" ]; then
        while IFS= read -r endpoint; do
            health_endpoints+=("$endpoint")
        done < "$CONFIG_DIR/${env}_health_checks.txt"
    fi

    # Default health checks
    if [ ${#health_endpoints[@]} -eq 0 ]; then
        health_endpoints=("http://localhost:8080/health" "http://localhost:3000/health")
    fi

    local failed_checks=0

    for endpoint in "${health_endpoints[@]}; do
        log "Checking health endpoint: $endpoint"

        if curl -f -s "$endpoint" >/dev/null; then
            log "Health check passed: $endpoint"
        else
            log "Health check failed: $endpoint"
            ((failed_checks++))
        fi
    done
}

```

```

    if [ $failed_checks -eq 0 ]; then
        log "All health checks passed"
        return 0
    else
        log "ERROR: $failed_checks health checks failed"
        return 1
    fi
}

# Generate environment documentation
generate_documentation() {
    local env="$1"
    local doc_file="docs/${env}_environment.md"

    log "Generating documentation for environment: $env"

    mkdir -p docs

    {
        echo "# $env Environment Documentation"
        echo
        echo "Generated on: $(date)"
        echo

        echo "## Infrastructure"
        if [ -f "$CONFIG_DIR/${env}_outputs.json" ]; then
            echo "### Terraform Outputs"
            echo '`json`'
            cat "$CONFIG_DIR/${env}_outputs.json"
            echo '``'
        fi

        echo
        echo "## Services"
        if command -v kubectl >/dev/null 2>&1; then
            echo "### Kubernetes Services"
            echo '``'
            kubectl get services -n "$KUBERNETES_NAMESPACE" 2>/dev/null || echo "No Kubernetes"
            echo '``'
        fi

        echo
        echo "## Configuration"
        if [ -f "$CONFIG_DIR/${env}.conf" ]; then
            echo "### Environment Configuration"
            echo '`bash`'
            cat "$CONFIG_DIR/${env}.conf"
        fi
    }
}

```

```

        echo '```'
    fi

} > "$doc_file"

log "Documentation generated: $doc_file"
}

# Cleanup environment
cleanup_environment() {
    local env="$1"

    log "Cleaning up environment: $env"

    # Confirm cleanup
    read -p "Are you sure you want to destroy the $env environment? (yes/no): " confirm
    if [ "$confirm" != "yes" ]; then
        log "Cleanup cancelled"
        return 0
    fi

    # Destroy Kubernetes resources
    if [ -d "k8s/$env" ] && command -v kubectl >/dev/null 2>&1; then
        kubectl delete -f "k8s/$env/" -n "$KUBERNETES_NAMESPACE" 2>/dev/null || true
        log "Kubernetes resources deleted"
    fi

    # Destroy Docker Compose services
    if [ -f "docker-compose.${env}.yaml" ]; then
        docker-compose -f "docker-compose.${env}.yaml" down -v
        log "Docker Compose services stopped"
    fi

    # Destroy Terraform infrastructure
    if [ -d "$TERRAFORM_DIR" ]; then
        cd "$TERRAFORM_DIR"
        terraform workspace select "$env"
        terraform destroy -var-file="$CONFIG_DIR/${env}.tfvars" -auto-approve
        terraform workspace select default
        terraform workspace delete "$env"
        cd - >/dev/null
        log "Infrastructure destroyed"
    fi

    log "Environment cleanup completed"
}

```

```

# Show environment status
show_status() {
    local env="$1"

    echo "Environment Status: $env"
    echo "======"
    echo

    # Terraform status
    if [ -d "$TERRAFORM_DIR" ]; then
        echo "Infrastructure Status:"
        cd "$TERRAFORM_DIR"
        terraform workspace select "$env" 2>/dev/null && terraform show -json | jq -r '.values'
        cd - >/dev/null
        echo
    fi

    # Kubernetes status
    if command -v kubectl >/dev/null 2>&1; then
        echo "Kubernetes Status:"
        kubectl get all -n "$KUBERNETES_NAMESPACE" 2>/dev/null || echo "No Kubernetes resources"
        echo
    fi

    # Docker status
    if [ -f "docker-compose.${env}.yaml" ]; then
        echo "Docker Compose Status:"
        docker-compose -f "docker-compose.${env}.yaml" ps
    fi
}

# Main provisioning process
provision_environment() {
    local env="$1"

    log "Starting environment provisioning: $env"

    check_prerequisites
    load_environment_config "$env"

    # Provision infrastructure
    if provision_infrastructure "$env"; then
        log "Infrastructure provisioning completed"
    else
        log "ERROR: Infrastructure provisioning failed"
        exit 1
    fi
}

```

```

# Configure servers
if configure_servers "$env"; then
    log "Server configuration completed"
else
    log "ERROR: Server configuration failed"
    exit 1
fi

# Deploy applications
if deploy_applications "$env"; then
    log "Application deployment completed"
else
    log "ERROR: Application deployment failed"
    exit 1
fi

# Setup monitoring
setup_monitoring "$env"

# Run health checks
if run_health_checks "$env"; then
    log "Health checks passed"
else
    log "WARNING: Some health checks failed"
fi

# Generate documentation
generate_documentation "$env"

log "Environment provisioning completed successfully: $env"
}

# Main execution
case "${2:-provision}" in
    provision)
        provision_environment "$ENVIRONMENT"
        ;;
    infrastructure)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        provision_infrastructure "$ENVIRONMENT"
        ;;
    configure)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        configure_servers "$ENVIRONMENT"

```

```

        ;;
    deploy)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        deploy_applications "$ENVIRONMENT"
        ;;
    health)
        load_environment_config "$ENVIRONMENT"
        run_health_checks "$ENVIRONMENT"
        ;;
    status)
        show_status "$ENVIRONMENT"
        ;;
    cleanup)
        cleanup_environment "$ENVIRONMENT"
        ;;
    docs)
        generate_documentation "$ENVIRONMENT"
        ;;
    *)
        echo "Usage: $0 <environment> {provision|infrastructure|configure|deploy|health|status}"
        echo
        echo "Environments: ${!ENV_CONFIGS[*]}"
        echo
        echo "Commands:"
        echo "  provision      - Full environment provisioning"
        echo "  infrastructure - Provision infrastructure only"
        echo "  configure      - Configure servers only"
        echo "  deploy         - Deploy applications only"
        echo "  health         - Run health checks"
        echo "  status         - Show environment status"
        echo "  cleanup        - Destroy environment"
        echo "  docs           - Generate documentation"
        exit 1
        ;;
esac

```

Web Scrapping and API Automation

Web Scraper

```

#!/bin/bash
# web_scraper.sh - Web scrapping and data extraction tool

URL="$1"

```



```

OUTPUT_DIR="${OUTPUT_DIR:-./scraped_data}"
USER_AGENT="${USER_AGENT:-Mozilla/5.0 (Linux; Bash Web Scraper)}"
DELAY="${DELAY:-1}"
MAX_PAGES="${MAX_PAGES:-10}"

# Create output directory
mkdir -p "$OUTPUT_DIR"

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*" | tee -a "$OUTPUT_DIR/scraper.log"
}

# Download page with retry logic
download_page() {
    local url="$1"
    local output_file="$2"
    local max_retries=3
    local retry_count=0

    while [ $retry_count -lt $max_retries ]; do
        if curl -s -L -A "$USER_AGENT" \
            -H "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8" \
            -H "Accept-Language: en-US,en;q=0.5" \
            -H "Accept-Encoding: gzip, deflate" \
            -H "Connection: keep-alive" \
            --compressed \
            "$url" > "$output_file"; then
            log "Downloaded: $url"
            return 0
        else
            ((retry_count++))
            log "Retry $retry_count/$max_retries for: $url"
            sleep $((retry_count * 2))
        fi
    done

    log "Failed to download after $max_retries retries: $url"
    return 1
}

# Extract links from HTML
extract_links() {
    local html_file="$1"
    local base_url="$2"

    # Extract all href attributes

```

```

grep -oP 'href="\K[^\"]*' "$html_file" | while read -r link; do
    # Convert relative URLs to absolute
    if [[ $link =~ ^https?:// ]]; then
        echo "$link"
    elif [[ $link =~ ^/ ]]; then
        echo "${base_url%}/$link"
    elif [[ $link =~ ^[/] ]]; then
        echo "${base_url%}/$link"
    fi
done | sort -u
}

# Extract text content
extract_text() {
    local html_file="$1"
    local output_file="$2"

    # Remove HTML tags and extract text
    sed 's/<[^>]*>//g' "$html_file" | \
    sed 's/ &nbsp; / /g; s/ & amp; / & /g; s/ & lt; /< /g; s/ & gt; /> /g; s/ & quot; /" /g' | \
    sed '/^[[:space:]]*$/d' > "$output_file"

    log "Extracted text to: $output_file"
}

# Extract specific data using patterns
extract_data() {
    local html_file="$1"
    local pattern="$2"
    local output_file="$3"

    grep -oP "$pattern" "$html_file" > "$output_file"
    local count=$(wc -l < "$output_file")
    log "Extracted $count items matching pattern: $pattern"
}

# Extract emails
extract_emails() {
    local html_file="$1"
    local output_file="$2"

    grep -oE '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' "$html_file" | sort -u > "$out
    local count=$(wc -l < "$output_file")
    log "Extracted $count unique email addresses"
}

# Extract phone numbers

```

```

extract_phones() {
    local html_file="$1"
    local output_file="$2"

    # Various phone number patterns
    grep -oE '(\+?[1-.\s]?)?(?([0-9]{3})?[-.\s]?[0-9]{3}[-.\s]?[0-9]{4})' "$html_file" | sort
    local count=$(wc -l < "$output_file")
    log "Extracted $count phone numbers"
}

# Extract images
extract_images() {
    local html_file="$1"
    local base_url="$2"
    local output_file="$3"

    grep -oP 'src="\K[^\"]*\.(jpg|jpeg|png|gif|webp)' "$html_file" | while read -r img_src; do
        # Convert relative URLs to absolute
        if [[ $img_src =~ ^https?:// ]]; then
            echo "$img_src"
        elif [[ $img_src =~ ^/ ]]; then
            echo "${base_url%}/$img_src"
        elif [[ $img_src =~ ^[/] ]]; then
            echo "${base_url%}/$img_src"
        fi
    done | sort -u > "$output_file"

    local count=$(wc -l < "$output_file")
    log "Extracted $count image URLs"
}

# Download images
download_images() {
    local image_list="$1"
    local download_dir="$OUTPUT_DIR/images"

    mkdir -p "$download_dir"

    while IFS= read -r img_url; do
        local filename=$(basename "$img_url" | cut -d'?' -f1)
        local output_path="$download_dir/$filename"

        if [ ! -f "$output_path" ]; then
            if curl -s -L -A "$USER_AGENT" "$img_url" -o "$output_path"; then
                log "Downloaded image: $filename"
            else

```

```

        log "Failed to download image: $img_url"
    fi
    sleep "$DELAY"
fi
done < "$image_list"
}

# Generate sitemap
generate_sitemap() {
    local links_file="$1"
    local sitemap_file="$OUTPUT_DIR/sitemap.xml"

    {
        echo '<?xml version="1.0" encoding="UTF-8"?>'
        echo '<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">'

        while IFS= read -r url; do
            echo "    <url>"
            echo "        <loc>$url</loc>"
            echo "        <lastmod>$(date -I)</lastmod>"
            echo "    </url>"
        done < "$links_file"

        echo '</urlset>'
    } > "$sitemap_file"

    log "Generated sitemap: $sitemap_file"
}

# Analyze page structure
analyze_structure() {
    local html_file="$1"
    local analysis_file="$OUTPUT_DIR/structure_analysis.txt"

    {
        echo "Page Structure Analysis"
        echo "======"
        echo "Generated: $(date)"
        echo

        echo "HTML Tags Count:"
        echo "======"
        grep -o '<[>]*>' "$html_file" | sed 's/<([^\>]*)\./\1/' | sort | uniq -c | sort -n

        echo
        echo "Meta Tags:"
        echo "======"
    }
}

```

```

    grep -i '<meta' "$html_file" || echo "No meta tags found"

    echo
    echo "Title:"
    echo "====="
    grep -oP '<title>\K[^\<]*' "$html_file" || echo "No title found"

    echo
    echo "Headings:"
    echo "====="
    grep -oP '<h[1-6][^\>]*\K[^\<]*' "$html_file" || echo "No headings found"

    echo
    echo "Forms:"
    echo "====="
    grep -c '<form' "$html_file" | xargs echo "Form count:"

    echo
    echo "Scripts:"
    echo "====="
    grep -c '<script' "$html_file" | xargs echo "Script count:"

    echo
    echo "Stylesheets:"
    echo "====="
    grep -c '<link.*stylesheet' "$html_file" | xargs echo "Stylesheet count:"

} > "$analysis_file"

log "Structure analysis saved: $analysis_file"
}

# Main scraping function
scrape_website() {
    local start_url="$1"
    local base_url=$(echo "$start_url" | sed 's|^(\(https\?:\/\/[^\/*\)\.|\1|')')

    log "Starting web scraping: $start_url"
    log "Base URL: $base_url"

    local page_count=0
    local visited_file="$OUTPUT_DIR/visited_urls.txt"
    local queue_file="$OUTPUT_DIR/url_queue.txt"

    # Initialize queue
    echo "$start_url" > "$queue_file"
    touch "$visited_file"

```

```

while [ $page_count -lt $MAX_PAGES ] && [ -s "$queue_file" ]; do
    # Get next URL from queue
    local current_url=$(head -1 "$queue_file")
    sed -i '1d' "$queue_file"

    # Skip if already visited
    if grep -Fxq "$current_url" "$visited_file"; then
        continue
    fi

    # Mark as visited
    echo "$current_url" >> "$visited_file"

    ((page_count++))
    log "Processing page $page_count/$MAX_PAGES: $current_url"

    # Download page
    local html_file="$OUTPUT_DIR/page_${page_count}.html"
    if download_page "$current_url" "$html_file"; then
        # Extract data
        extract_text "$html_file" "$OUTPUT_DIR/page_${page_count}.txt"
        extract_emails "$html_file" "$OUTPUT_DIR/page_${page_count}_emails.txt"
        extract_phones "$html_file" "$OUTPUT_DIR/page_${page_count}_phones.txt"
        extract_images "$html_file" "$base_url" "$OUTPUT_DIR/page_${page_count}_images.t

        # Extract links for further crawling
        local links_file="$OUTPUT_DIR/page_${page_count}_links.txt"
        extract_links "$html_file" "$base_url" > "$links_file"

        # Add new links to queue (same domain only)
        while IFS= read -r link; do
            if [[ $link =~ ^$base_url ]] && ! grep -Fxq "$link" "$visited_file"; then
                echo "$link" >> "$queue_file"
            fi
        done < "$links_file"

        # Analyze page structure for first page
        if [ $page_count -eq 1 ]; then
            analyze_structure "$html_file"
        fi
    fi

    # Respect rate limiting
    sleep "$DELAY"
done

```

```

log "Scraping completed. Processed $page_count pages."

# Generate consolidated reports
generate_reports
}

# Generate consolidated reports
generate_reports() {
    log "Generating consolidated reports..."

    # Consolidate all emails
    cat "$OUTPUT_DIR"/page_*_emails.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_emails.txt"
    local email_count=$(wc -l < "$OUTPUT_DIR/all_emails.txt" 2>/dev/null || echo 0)
    log "Total unique emails found: $email_count"

    # Consolidate all phone numbers
    cat "$OUTPUT_DIR"/page_*_phones.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_phones.txt"
    local phone_count=$(wc -l < "$OUTPUT_DIR/all_phones.txt" 2>/dev/null || echo 0)
    log "Total unique phone numbers found: $phone_count"

    # Consolidate all images
    cat "$OUTPUT_DIR"/page_*_images.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_images.txt"
    local image_count=$(wc -l < "$OUTPUT_DIR/all_images.txt" 2>/dev/null || echo 0)
    log "Total unique images found: $image_count"

    # Consolidate all links
    cat "$OUTPUT_DIR"/page_*_links.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_links.txt"
    local link_count=$(wc -l < "$OUTPUT_DIR/all_links.txt" 2>/dev/null || echo 0)
    log "Total unique links found: $link_count"

    # Generate sitemap
    if [ -f "$OUTPUT_DIR/all_links.txt" ]; then
        generate_sitemap "$OUTPUT_DIR/all_links.txt"
    fi

    # Generate summary report
    {
        echo "Web Scraping Summary Report"
        echo "======"
        echo "Generated: $(date)"
        echo "Target URL: $URL"
        echo "Pages processed: $(ls "$OUTPUT_DIR"/page_*.html 2>/dev/null | wc -l)"
        echo "Unique emails: $email_count"
        echo "Unique phones: $phone_count"
        echo "Unique images: $image_count"
        echo "Unique links: $link_count"
        echo
    }
}

```

```

        echo "Output directory: $OUTPUT_DIR"
        echo
        echo "Files generated:"
        ls -la "$OUTPUT_DIR"
    } > "$OUTPUT_DIR/summary_report.txt"

    log "Summary report generated: $OUTPUT_DIR/summary_report.txt"
}

# Show usage
show_usage() {
    cat << EOF
Web Scraper - Extract data from websites

Usage: $0 <url> [options]

Options:
    -d, --delay SECONDS      Delay between requests (default: 1)
    -m, --max-pages NUM      Maximum pages to scrape (default: 10)
    -o, --output DIR         Output directory (default: ./scraped_data)
    -u, --user-agent STR     User agent string
    -i, --images             Download images
    -h, --help               Show this help

Environment Variables:
    OUTPUT_DIR               Output directory
    USER_AGENT              User agent string
    DELAY                    Delay between requests
    MAX_PAGES                Maximum pages to scrape

Examples:
    $0 https://example.com
    $0 https://example.com -d 2 -m 5
    $0 https://example.com --images
EOF
}

# Parse command line arguments
DOWNLOAD_IMAGES=false

while [[ $# -gt 0 ]]; do
    case $1 in
        -d|--delay)
            DELAY="$2"
            shift 2
            ;;

```



```

-m|--max-pages)
    MAX_PAGES="$2"
    shift 2
    ;;
-o|--output)
    OUTPUT_DIR="$2"
    shift 2
    ;;
-u|--user-agent)
    USER_AGENT="$2"
    shift 2
    ;;
-i|--images)
    DOWNLOAD_IMAGES=true
    shift
    ;;
-h|--help)
    show_usage
    exit 0
    ;;
-*)
    echo "Unknown option: $1"
    show_usage
    exit 1
    ;;
*)
    if [ -z "$URL" ]; then
        URL="$1"
    fi
    shift
    ;;
esac
done

# Main execution
if [ -z "$URL" ]; then
    echo "Error: URL is required"
    show_usage
    exit 1
fi

# Validate URL
if ! [[ $URL =~ ^https?:// ]]; then
    echo "Error: Invalid URL format. Must start with http:// or https://"
    exit 1
fi

```

```

# Check dependencies
if ! command -v curl >/dev/null 2>&1; then
    echo "Error: curl is required but not installed"
    exit 1
fi

# Start scraping
scrape_website "$URL"

# Download images if requested
if [ "$DOWNLOAD_IMAGES" = true ] && [ -f "$OUTPUT_DIR/all_images.txt" ]; then
    log "Downloading images..."
    download_images "$OUTPUT_DIR/all_images.txt"
fi

log "Web scraping completed successfully"
echo "Results saved in: $OUTPUT_DIR"

```

These automation examples demonstrate comprehensive solutions for common DevOps and system administration tasks. Each script includes error handling, logging, configuration management, and notification capabilities, making them suitable for production environments.

Utility Scripts

This section contains practical utility scripts for everyday tasks, system maintenance, and productivity enhancement.

File and Directory Utilities

Duplicate File Finder

```
#!/bin/bash
# find_duplicates.sh - Find and manage duplicate files

SEARCH_DIR="${1:-.}"
ACTION="${ACTION:-list}"
MIN_SIZE="${MIN_SIZE:-1}"
HASH_ALGORITHM="${HASH_ALGORITHM:-md5sum}"

# Temporary files
TEMP_DIR="/tmp/duplicate_finder.$$"
mkdir -p "$TEMP_DIR"

# Cleanup function
cleanup() {
    rm -rf "$TEMP_DIR"
}
trap cleanup EXIT

# Logging function
log() {
    echo "$(date '+%H:%M:%S') - $*"
}

# Calculate file hash
calculate_hash() {
    local file="$1"
    case "$HASH_ALGORITHM" in
        md5sum) md5sum "$file" | cut -d' ' -f1 ;;
        sha1sum) sha1sum "$file" | cut -d' ' -f1 ;;
        sha256sum) sha256sum "$file" | cut -d' ' -f1 ;;
        *) md5sum "$file" | cut -d' ' -f1 ;;
    esac
}
```

```

    esac
}

# Find files by size
find_by_size() {
    log "Finding files by size..."

    find "$SEARCH_DIR" -type f -size +${MIN_SIZE}c -exec stat -c "%s %n" {} \; | \
    sort -n > "$TEMP_DIR/files_by_size.txt"

    # Group files by size
    awk '{print $1}' "$TEMP_DIR/files_by_size.txt" | uniq -d > "$TEMP_DIR/duplicate_sizes.txt"

    local duplicate_size_count=$(wc -l < "$TEMP_DIR/duplicate_sizes.txt")
    log "Found $duplicate_size_count different file sizes with potential duplicates"
}

# Find duplicates by hash
find_duplicates() {
    log "Calculating file hashes..."

    local processed=0
    local total=$(wc -l < "$TEMP_DIR/duplicate_sizes.txt")

    while IFS= read -r size; do
        ((processed++))
        echo -ne "\rProgress: $processed/$total"

        # Get all files of this size
        grep "^$size " "$TEMP_DIR/files_by_size.txt" | cut -d' ' -f2- | while IFS= read -r file; do
            if [ -f "$file" ]; then
                local hash=$(calculate_hash "$file")
                echo "$hash $size $file"
            fi
        done >> "$TEMP_DIR/files_with_hash.txt"
    done < "$TEMP_DIR/duplicate_sizes.txt"

    echo

    # Group by hash
    sort "$TEMP_DIR/files_with_hash.txt" > "$TEMP_DIR/sorted_hashes.txt"
    awk '{print $1}' "$TEMP_DIR/sorted_hashes.txt" | uniq -d > "$TEMP_DIR/duplicate_hashes.txt"

    local duplicate_hash_count=$(wc -l < "$TEMP_DIR/duplicate_hashes.txt")
    log "Found $duplicate_hash_count groups of duplicate files"
}

```

```

# List duplicate files
list_duplicates() {
    log "Listing duplicate files..."

    local group_num=0
    local total_duplicates=0
    local total_wasted_space=0

    while IFS= read -r hash; do
        ((group_num++))
        echo
        echo "Duplicate Group $group_num (Hash: $hash):"
        echo "===== "

        local files=()
        local file_count=0
        local file_size=0

        grep "^$hash " "$TEMP_DIR/sorted_hashes.txt" | while IFS=' ' read -r h size file; do
            ((file_count++))
            file_size="$size"
            echo "$file_count. $file ($(numfmt --to=iec-i --suffix=B $size))"
            files+=("$file")
        done

        # Calculate wasted space (all but one file)
        local wasted=$((file_size * (file_count - 1)))
        total_wasted_space=$((total_wasted_space + wasted))
        total_duplicates=$((total_duplicates + file_count - 1))

        echo "Files in group: $file_count"
        echo "Wasted space: $(numfmt --to=iec-i --suffix=B $wasted)"

    done < "$TEMP_DIR/duplicate_hashes.txt"

    echo
    echo "Summary:"
    echo "===== "
    echo "Duplicate groups: $group_num"
    echo "Total duplicate files: $total_duplicates"
    echo "Total wasted space: $(numfmt --to=iec-i --suffix=B $total_wasted_space)"
}

# Interactive duplicate removal
interactive_remove() {
    log "Starting interactive duplicate removal..."

```

```

local group_num=0
local removed_count=0
local freed_space=0

while IFS= read -r hash; do
    ((group_num++))
    echo
    echo "Duplicate Group $group_num (Hash: $hash):"
    echo "===== "

    local files=()
    local file_count=0
    local file_size=0

    # Collect files in this group
    while IFS=' ' read -r h size file; do
        if [ "$h" = "$hash" ]; then
            ((file_count++))
            file_size="$size"
            files+=("$file")
            echo "$file_count. $file"
            echo "    Size: $(numfmt --to=iec-i --suffix=B $size)"
            echo "    Modified: $(stat -c %y "$file")"
        fi
    done < "$TEMP_DIR/sorted_hashes.txt"

    echo
    echo "Choose action:"
    echo "k) Keep all files"
    echo "1-$file_count) Keep only file number X (delete others)"
    echo "a) Auto-keep oldest file"
    echo "s) Skip this group"

    read -p "Your choice: " choice

    case "$choice" in
        k|K)
            echo "Keeping all files in this group"
            ;;
        [1-9]*)
            if [ "$choice" -ge 1 ] && [ "$choice" -le $file_count ]; then
                local keep_index=$((choice - 1))
                echo "Keeping: ${files[$keep_index]}"

                for i in "${!files[@]}"; do
                    if [ $i -ne $keep_index ]; then
                        echo "Removing: ${files[$i]}"
                    fi
                done
            fi
        *)
            :
    esac
done

```

```

        if rm "${files[$i]}"; then
            ((removed_count++))
            freed_space=$((freed_space + file_size))
        fi
    fi
done
else
    echo "Invalid choice, skipping group"
fi
;;
a|A)
    # Keep oldest file
    local oldest_file=""
    local oldest_time=0

    for file in "${files[@]}"; do
        local mtime=$(stat -c %Y "$file")
        if [ $oldest_time -eq 0 ] || [ $mtime -lt $oldest_time ]; then
            oldest_time=$mtime
            oldest_file="$file"
        fi
    done

    echo "Auto-keeping oldest file: $oldest_file"

    for file in "${files[@]}"; do
        if [ "$file" != "$oldest_file" ]; then
            echo "Removing: $file"
            if rm "$file"; then
                ((removed_count++))
                freed_space=$((freed_space + file_size))
            fi
        fi
    done
    ;;
s|S)
    echo "Skipping this group"
    ;;
*)
    echo "Invalid choice, skipping group"
    ;;
esac

done < "$TEMP_DIR/duplicate_hashes.txt"

echo

```

```

    echo "Removal Summary:"
    echo "======"
    echo "Files removed: $removed_count"
    echo "Space freed: $(numfmt --to=iec-i --suffix=B $freed_space)"
}

# Auto-remove duplicates (keep oldest)
auto_remove() {
    log "Auto-removing duplicates (keeping oldest files)..."

    local removed_count=0
    local freed_space=0

    while IFS= read -r hash; do
        local files=()
        local file_size=0

        # Collect files in this group
        while IFS=' ' read -r h size file; do
            if [ "$h" = "$hash" ]; then
                files+=("$file")
                file_size="$size"
            fi
        done < "$TEMP_DIR/sorted_hashes.txt"

        # Find oldest file
        local oldest_file=""
        local oldest_time=0

        for file in "${files[@]}; do
            local mtime=$(stat -c %Y "$file")
            if [ $oldest_time -eq 0 ] || [ $mtime -lt $oldest_time ]; then
                oldest_time=$mtime
                oldest_file="$file"
            fi
        done

        echo "Keeping oldest: $oldest_file"

        # Remove other files
        for file in "${files[@]}; do
            if [ "$file" != "$oldest_file" ]; then
                echo "Removing: $file"
                if rm "$file"; then
                    ((removed_count++))
                    freed_space=$((freed_space + file_size))
                fi
            fi
        done
    done
}

```



```

        fi
    done

done < "$TEMP_DIR/duplicate_hashes.txt"

echo
echo "Auto-removal Summary:"
echo "======"
echo "Files removed: $removed_count"
echo "Space freed: $(numfmt --to=iec-i --suffix=B $freed_space)"
}

# Generate report
generate_report() {
    local report_file="duplicate_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "Duplicate Files Report"
        echo "======"
        echo "Generated: $(date)"
        echo "Search directory: $SEARCH_DIR"
        echo "Minimum file size: $MIN_SIZE bytes"
        echo "Hash algorithm: $HASH_ALGORITHM"
        echo

        list_duplicates

    } > "$report_file"

    log "Report saved: $report_file"
}

# Main execution
main() {
    if [ ! -d "$SEARCH_DIR" ]; then
        echo "Error: Directory '$SEARCH_DIR' does not exist"
        exit 1
    fi

    log "Starting duplicate file search in: $SEARCH_DIR"
    log "Minimum file size: $MIN_SIZE bytes"
    log "Hash algorithm: $HASH_ALGORITHM"

    find_by_size
    find_duplicates

    case "$ACTION" in

```

```

        list)
            list_duplicates
            ;;
        interactive)
            interactive_remove
            ;;
        auto)
            auto_remove
            ;;
        report)
            generate_report
            ;;
        *)
            echo "Unknown action: $ACTION"
            echo "Available actions: list, interactive, auto, report"
            exit 1
            ;;
    esac
}

# Show usage
show_usage() {
    cat << EOF
Duplicate File Finder - Find and manage duplicate files

Usage: $0 [directory] [options]

Options:
    -a, --action ACTION      Action to perform: list, interactive, auto, report
    -s, --min-size SIZE      Minimum file size in bytes (default: 1)
    -h, --hash ALGORITHM     Hash algorithm: md5sum, sha1sum, sha256sum
    --help                   Show this help

Environment Variables:
    ACTION                   Action to perform
    MIN_SIZE                 Minimum file size
    HASH_ALGORITHM           Hash algorithm

Actions:
    list                    List duplicate files (default)
    interactive             Interactive duplicate removal
    auto                    Auto-remove duplicates (keep oldest)
    report                  Generate detailed report

Examples:
    $0 /home/user/Documents
    $0 /home/user/Pictures --action interactive

```

```

    ACTION=auto MIN_SIZE=1024 $0 /var/log
EOF
}

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    case $1 in
        -a|--action)
            ACTION="$2"
            shift 2
            ;;
        -s|--min-size)
            MIN_SIZE="$2"
            shift 2
            ;;
        -h|--hash)
            HASH_ALGORITHM="$2"
            shift 2
            ;;
        --help)
            show_usage
            exit 0
            ;;
        -* )
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            if [ -z "$SEARCH_DIR" ] || [ "$SEARCH_DIR" = "." ]; then
                SEARCH_DIR="$1"
            fi
            shift
            ;;
    esac
done

main

```

Directory Synchronizer

```

#!/bin/bash
# sync_directories.sh - Synchronize directories with various options

SOURCE_DIR="$1"

```

```

TARGET_DIR="$2"
SYNC_MODE="${SYNC_MODE:-mirror}"
DRY_RUN="${DRY_RUN:-false}"
EXCLUDE_FILE="${EXCLUDE_FILE:-}"
LOG_FILE="${LOG_FILE:-sync_$(date +%Y%m%d_%H%M%S).log}"

# Sync statistics
declare -i files_copied=0
declare -i files_updated=0
declare -i files_deleted=0
declare -i dirs_created=0
declare -i bytes_transferred=0

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Check if file should be excluded
is_excluded() {
    local file="$1"

    if [ -n "$EXCLUDE_FILE" ] && [ -f "$EXCLUDE_FILE" ]; then
        while IFS= read -r pattern; do
            # Skip comments and empty lines
            [[ $pattern =~ ^[:space:]*$# ]] && continue
            [[ -z $pattern ]] && continue

            if [[ $file =~ $pattern ]]; then
                return 0 # File is excluded
            fi
        done < "$EXCLUDE_FILE"
    fi

    return 1 # File is not excluded
}

# Copy file with verification
copy_file() {
    local src="$1"
    local dst="$2"
    local action="$3" # "copy" or "update"

    if [ "$DRY_RUN" = "true" ]; then

```

```

        log INFO "DRY RUN: Would $action $src -> $dst"
        return 0
    fi

    # Create destination directory if needed
    local dst_dir=$(dirname "$dst")
    if [ ! -d "$dst_dir" ]; then
        if mkdir -p "$dst_dir"; then
            log INFO "Created directory: $dst_dir"
            ((dirs_created++))
        else
            log ERROR "Failed to create directory: $dst_dir"
            return 1
        fi
    fi

    # Copy file
    if cp -p "$src" "$dst"; then
        local file_size=$(stat -c%s "$src" 2>/dev/null || stat -f%z "$src" 2>/dev/null)
        bytes_transferred=$((bytes_transferred + file_size))

        if [ "$action" = "copy" ]; then
            log INFO "Copied: $src -> $dst ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((files_copied++))
        else
            log INFO "Updated: $src -> $dst ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((files_updated++))
        fi
        return 0
    else
        log ERROR "Failed to $action: $src -> $dst"
        return 1
    fi
}

# Delete file or directory
delete_item() {
    local item="$1"

    if [ "$DRY_RUN" = "true" ]; then
        log INFO "DRY RUN: Would delete $item"
        return 0
    fi

    if rm -rf "$item"; then
        log INFO "Deleted: $item"
        ((files_deleted++))
    fi
}

```

```

        return 0
    else
        log ERROR "Failed to delete: $item"
        return 1
    fi
}

# Compare files
files_different() {
    local src="$1"
    local dst="$2"

    # Check if destination exists
    [ ! -f "$dst" ] && return 0

    # Compare modification times
    local src_mtime=$(stat -c %Y "$src" 2>/dev/null || stat -f %m "$src" 2>/dev/null)
    local dst_mtime=$(stat -c %Y "$dst" 2>/dev/null || stat -f %m "$dst" 2>/dev/null)

    [ "$src_mtime" -gt "$dst_mtime" ] && return 0

    # Compare sizes
    local src_size=$(stat -c %s "$src" 2>/dev/null || stat -f %z "$src" 2>/dev/null)
    local dst_size=$(stat -c %s "$dst" 2>/dev/null || stat -f %z "$dst" 2>/dev/null)

    [ "$src_size" -ne "$dst_size" ] && return 0

    return 1 # Files are the same
}

# Mirror sync (make target identical to source)
sync_mirror() {
    log INFO "Starting mirror synchronization"
    log INFO "Source: $SOURCE_DIR"
    log INFO "Target: $TARGET_DIR"

    # First pass: copy/update files from source to target
    find "$SOURCE_DIR" -type f | while IFS= read -r src_file; do
        # Skip excluded files
        if is_excluded "$src_file"; then
            log DEBUG "Excluded: $src_file"
            continue
        fi

        # Calculate relative path
        local rel_path="${src_file#$SOURCE_DIR/}"
        local dst_file="$TARGET_DIR/$rel_path"

```

```

        if files_different "$src_file" "$dst_file"; then
            if [ -f "$dst_file" ]; then
                copy_file "$src_file" "$dst_file" "update"
            else
                copy_file "$src_file" "$dst_file" "copy"
            fi
        fi
    done

# Second pass: remove files from target that don't exist in source
if [ -d "$TARGET_DIR" ]; then
    find "$TARGET_DIR" -type f | while IFS= read -r dst_file; do
        local rel_path="${dst_file#$TARGET_DIR/}"
        local src_file="$SOURCE_DIR/$rel_path"

        if [ ! -f "$src_file" ]; then
            delete_item "$dst_file"
        fi
    done

# Remove empty directories
    find "$TARGET_DIR" -type d -empty -delete 2>/dev/null || true
fi
}

# One-way sync (only copy new/updated files)
sync_oneway() {
    log INFO "Starting one-way synchronization"
    log INFO "Source: $SOURCE_DIR"
    log INFO "Target: $TARGET_DIR"

    find "$SOURCE_DIR" -type f | while IFS= read -r src_file; do
        # Skip excluded files
        if is_excluded "$src_file"; then
            log DEBUG "Excluded: $src_file"
            continue
        fi

        # Calculate relative path
        local rel_path="${src_file#$SOURCE_DIR/}"
        local dst_file="$TARGET_DIR/$rel_path"

        if files_different "$src_file" "$dst_file"; then
            if [ -f "$dst_file" ]; then
                copy_file "$src_file" "$dst_file" "update"
            else

```

```

        copy_file "$src_file" "$dst_file" "copy"
    fi
fi
done
}

# Bidirectional sync (sync both ways based on modification time)
sync_bidirectional() {
    log INFO "Starting bidirectional synchronization"
    log INFO "Directory A: $SOURCE_DIR"
    log INFO "Directory B: $TARGET_DIR"

    # Create list of all files in both directories
    local all_files_file="/tmp/sync_all_files.$$"

    {
        find "$SOURCE_DIR" -type f | sed "s|^$SOURCE_DIR/||"
        find "$TARGET_DIR" -type f | sed "s|^$TARGET_DIR/||"
    } | sort -u > "$all_files_file"

    while IFS= read -r rel_path; do
        # Skip excluded files
        if is_excluded "$rel_path"; then
            log DEBUG "Excluded: $rel_path"
            continue
        fi

        local file_a="$SOURCE_DIR/$rel_path"
        local file_b="$TARGET_DIR/$rel_path"

        if [ -f "$file_a" ] && [ -f "$file_b" ]; then
            # Both files exist, sync newer to older
            local mtime_a=$(stat -c %Y "$file_a" 2>/dev/null || stat -f %m "$file_a" 2>/dev/null)
            local mtime_b=$(stat -c %Y "$file_b" 2>/dev/null || stat -f %m "$file_b" 2>/dev/null)

            if [ "$mtime_a" -gt "$mtime_b" ]; then
                copy_file "$file_a" "$file_b" "update"
            elif [ "$mtime_b" -gt "$mtime_a" ]; then
                copy_file "$file_b" "$file_a" "update"
            fi
        elif [ -f "$file_a" ]; then
            # File only exists in A, copy to B
            copy_file "$file_a" "$file_b" "copy"
        elif [ -f "$file_b" ]; then
            # File only exists in B, copy to A
            copy_file "$file_b" "$file_a" "copy"
        fi
    done
}

```



```

done < "$all_files_file"

rm -f "$all_files_file"
}

# Generate sync report
generate_report() {
    local report_file="sync_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "Directory Synchronization Report"
        echo "======"
        echo "Generated: $(date)"
        echo "Source: $SOURCE_DIR"
        echo "Target: $TARGET_DIR"
        echo "Sync Mode: $SYNC_MODE"
        echo "Dry Run: $DRY_RUN"
        echo

        echo "Statistics:"
        echo "======"
        echo "Files copied: $files_copied"
        echo "Files updated: $files_updated"
        echo "Files deleted: $files_deleted"
        echo "Directories created: $dirs_created"
        echo "Bytes transferred: $(numfmt --to=iec-i --suffix=B $bytes_transferred)"
        echo

        echo "Log file: $LOG_FILE"

        if [ -f "$EXCLUDE_FILE" ]; then
            echo
            echo "Exclusion patterns:"
            echo "======"
            cat "$EXCLUDE_FILE"
        fi

    } > "$report_file"

    log INFO "Report generated: $report_file"
}

# Validate directories
validate_directories() {
    if [ ! -d "$SOURCE_DIR" ]; then
        log ERROR "Source directory does not exist: $SOURCE_DIR"
        exit 1
    fi
}

```

```

fi

if [ "$SYNC_MODE" != "bidirectional" ] && [ ! -d "$TARGET_DIR" ]; then
    log INFO "Creating target directory: $TARGET_DIR"
    if ! mkdir -p "$TARGET_DIR"; then
        log ERROR "Failed to create target directory: $TARGET_DIR"
        exit 1
    fi
fi

# Check for circular sync (target inside source or vice versa)
local source_real=$(realpath "$SOURCE_DIR")
local target_real=$(realpath "$TARGET_DIR" 2>/dev/null || echo "$TARGET_DIR")

if [[ "$target_real" == "$source_real"* ]] || [[ "$source_real" == "$target_real"* ]]; then
    log ERROR "Circular sync detected: target cannot be inside source or vice versa"
    exit 1
fi
}

# Show progress
show_progress() {
    if [ "$DRY_RUN" != "true" ]; then
        while true; do
            echo -ne "\rFiles: $files_copied copied, $files_updated updated, $files_deleted deleted"
            sleep 1
        done &
        local progress_pid=$!

        # Kill progress display when script exits
        trap "kill $progress_pid 2>/dev/null" EXIT
    fi
}

# Main synchronization function
main() {
    log INFO "Starting directory synchronization"

    validate_directories

    if [ "$DRY_RUN" = "true" ]; then
        log INFO "DRY RUN MODE - No files will be modified"
    fi

    show_progress

    case "$SYNC_MODE" in

```

```

        mirror)
            sync_mirror
            ;;
        oneway)
            sync_oneway
            ;;
        bidirectional)
            sync_bidirectional
            ;;
        *)
            log ERROR "Unknown sync mode: $SYNC_MODE"
            exit 1
            ;;
    esac

    log INFO "Synchronization completed"

    echo
    echo "Synchronization Summary:"
    echo "======"
    echo "Files copied: $files_copied"
    echo "Files updated: $files_updated"
    echo "Files deleted: $files_deleted"
    echo "Directories created: $dirs_created"
    echo "Bytes transferred: $(numfmt --to=iec-i --suffix=B $bytes_transferred)"

    generate_report
}

# Show usage
show_usage() {
    cat << EOF
Directory Synchronizer - Synchronize directories with various options

Usage: $0 <source_dir> <target_dir> [options]

Options:
-m, --mode MODE           Sync mode: mirror, oneway, bidirectional
-n, --dry-run             Show what would be done without making changes
-e, --exclude FILE        File containing exclusion patterns
-l, --log FILE            Log file path
--help                   Show this help

Environment Variables:
SYNC_MODE                Sync mode
DRY_RUN                  Dry run mode (true/false)
EXCLUDE_FILE             Exclusion patterns file

```

LOG_FILE Log file path

Sync Modes:

mirror	Make target identical to source (default)
oneway	Copy new/updated files from source to target
bidirectional	Sync both ways based on modification time

Examples:

```
$0 /home/user/docs /backup/docs
$0 /home/user/docs /backup/docs --mode oneway --dry-run
SYNC_MODE=bidirectional $0 /home/user/docs /shared/docs
```

EOF

}

```
# Parse command line arguments
while [[ $# -gt 0 ]]; do
    case $1 in
        -m|--mode)
            SYNC_MODE="$2"
            shift 2
            ;;
        -n|--dry-run)
            DRY_RUN="true"
            shift
            ;;
        -e|--exclude)
            EXCLUDE_FILE="$2"
            shift 2
            ;;
        -l|--log)
            LOG_FILE="$2"
            shift 2
            ;;
        --help)
            show_usage
            exit 0
            ;;
        -* )
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            if [ -z "$SOURCE_DIR" ]; then
                SOURCE_DIR="$1"
            elif [ -z "$TARGET_DIR" ]; then
                TARGET_DIR="$1"
            fi
    esac
done
```

```

        fi
        shift
        ;;
    esac
done

# Validate required arguments
if [ -z "$SOURCE_DIR" ] || [ -z "$TARGET_DIR" ]; then
    echo "Error: Both source and target directories are required"
    show_usage
    exit 1
fi

main

```

System Maintenance Utilities

Log Rotator

```

#!/bin/bash
# log_rotator.sh - Advanced log rotation utility

CONFIG_FILE="${CONFIG_FILE:-/etc/logrotate_custom.conf}"
LOG_DIR="${LOG_DIR:-/var/log}"
MAX_AGE_DAYS="${MAX_AGE_DAYS:-30}"
COMPRESS="${COMPRESS:-true}"
COMPRESSION_DELAY="${COMPRESSION_DELAY:-1}"

# Default configuration
declare -A default_config=(
    [rotate]="7"
    [size]="100M"
    [compress]="true"
    [delaycompress]="true"
    [missingok]="true"
    [notifempty]="true"
    [create]="644 root root"
)

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*" | tee -a "/var/log/log_rotator.log"
}

```

```

# Parse size string to bytes
parse_size() {
    local size_str="$1"
    local size_num=$(echo "$size_str" | sed 's/[^0-9]*//g')
    local size_unit=$(echo "$size_str" | sed 's/[0-9]*//g' | tr '[:lower:]' '[:upper:]')

    case "$size_unit" in
        K|KB) echo $((size_num * 1024)) ;;
        M|MB) echo $((size_num * 1024 * 1024)) ;;
        G|GB) echo $((size_num * 1024 * 1024 * 1024)) ;;
        *) echo "$size_num" ;;
    esac
}

# Check if file needs rotation
needs_rotation() {
    local log_file="$1"
    local max_size="$2"
    local rotate_count="$3"

    # Check if file exists and is not empty
    if [ ! -f "$log_file" ] || [ ! -s "$log_file" ]; then
        return 1
    fi

    # Check size
    local file_size=$(stat -c%s "$log_file" 2>/dev/null || stat -f%z "$log_file" 2>/dev/null)
    local max_bytes=$(parse_size "$max_size")

    if [ "$file_size" -gt "$max_bytes" ]; then
        return 0
    fi

    # Check if we have reached max rotations
    if [ -f "${log_file}.${rotate_count}" ]; then
        return 0
    fi

    return 1
}

# Rotate a single log file
rotate_log() {
    local log_file="$1"
    local config_name="$2"

    # Load configuration for this log

```

```

local rotate_count="${log_configs[$config_name.rotate]:-${default_config[rotate]}}"
local max_size="${log_configs[$config_name.size]:-${default_config[size]}}"
local compress="${log_configs[$config_name.compress]:-${default_config[compress]}}"
local delaycompress="${log_configs[$config_name.delaycompress]:-${default_config[delayco
local missingok="${log_configs[$config_name.missingok]:-${default_config[missingok]}}"
local notifempty="${log_configs[$config_name.notifempty]:-${default_config[notifempty]}}"
local create_mode="${log_configs[$config_name.create]:-${default_config[create]}}"

# Check if rotation is needed
if ! needs_rotation "$log_file" "$max_size" "$rotate_count"; then
    log "No rotation needed for: $log_file"
    return 0
fi

log "Rotating log file: $log_file"

# Remove oldest log if it exists
if [ -f "${log_file}.${rotate_count}" ]; then
    rm "${log_file}.${rotate_count}"
    log "Removed oldest log: ${log_file}.${rotate_count}"
fi

# Rotate existing logs
for ((i=rotate_count-1; i>=1; i--)); do
    local current_log="${log_file}.${i}"
    local next_log="${log_file}.${(i+1)}"

    if [ -f "$current_log" ]; then
        mv "$current_log" "$next_log"
        log "Moved: $current_log -> $next_log"
    fi
done

# Move current log to .1
if [ -f "$log_file" ]; then
    mv "$log_file" "${log_file}.1"
    log "Moved: $log_file -> ${log_file}.1"

# Create new log file with proper permissions
if [ -n "$create_mode" ]; then
    local mode=$(echo "$create_mode" | awk '{print $1}')
    local owner=$(echo "$create_mode" | awk '{print $2}')
    local group=$(echo "$create_mode" | awk '{print $3}')

    touch "$log_file"
    chmod "$mode" "$log_file" 2>/dev/null || true

```

```

        chown "$owner:$group" "$log_file" 2>/dev/null || true

        log "Created new log file: $log_file ($mode $owner:$group)"
    fi
fi

# Compress rotated logs
if [ "$compress" = "true" ]; then
    local start_index=1
    if [ "$delaycompress" = "true" ]; then
        start_index=2
    fi

    for ((i=start_index; i<=rotate_count; i++)); do
        local log_to_compress="${log_file}.${i}"
        if [ -f "$log_to_compress" ] && [[ ! "$log_to_compress" =~ \.gz$ ]]; then
            if gzip "$log_to_compress"; then
                log "Compressed: $log_to_compress"
            else
                log "Failed to compress: $log_to_compress"
            fi
        fi
    done
fi

# Execute post-rotation script if configured
local postrotate_script="${log_configs[$config_name.postrotate]:-}"
if [ -n "$postrotate_script" ]; then
    log "Executing post-rotation script: $postrotate_script"
    if eval "$postrotate_script"; then
        log "Post-rotation script completed successfully"
    else
        log "Post-rotation script failed"
    fi
fi

}

# Load configuration file
load_config() {
    declare -gA log_configs

    if [ ! -f "$CONFIG_FILE" ]; then
        log "Configuration file not found: $CONFIG_FILE"
        return 1
    fi

    local current_section=""

```



```

while IFS= read -r line; do
    # Skip comments and empty lines
    [[ $line =~ ^[:space:]*# ]] && continue
    [[ -z $line ]] && continue

    # Check for section header
    if [[ $line =~ ^\[([^\]]+)\] ]]; then
        current_section="${BASH_REMATCH[1]}"
        continue
    fi

    # Parse key=value pairs
    if [[ $line =~ ^[:space:]*\[([^\]]+)\]([^\s=]+)=(.*)$ ]]; then
        local key="${BASH_REMATCH[1]// /}"
        local value="${BASH_REMATCH[2]}"

        if [ -n "$current_section" ]; then
            log_configs["$current_section.$key"]="$value"
        else
            log_configs["$key"]="$value"
        fi
    fi
done < "$CONFIG_FILE"

log "Configuration loaded from: $CONFIG_FILE"
}

# Generate default configuration
generate_config() {
    cat > "$CONFIG_FILE" << EOF
# Log Rotation Configuration
# Generated on $(date)

# Global settings
compress=true
delaycompress=true
missingok=true
notifempty=true

# Apache logs
[apache]
files=/var/log/apache2/*.log
rotate=52
size=100M
create=644 www-data www-data
postrotate=systemctl reload apache2

```

```

# Nginx logs
[nginx]
files=/var/log/nginx/*.log
rotate=52
size=100M
create=644 www-data www-data
postrotate=systemctl reload nginx

# System logs
[syslog]
files=/var/log/syslog
rotate=7
size=50M
create=644 syslog adm

# Application logs
[application]
files=/var/log/myapp/*.log
rotate=30
size=10M
create=644 myapp myapp
postrotate=killall -USR1 myapp
EOF

    log "Configuration template created: $CONFIG_FILE"
}

# Clean old compressed logs
cleanup_old_logs() {
    log "Cleaning up logs older than $MAX_AGE_DAYS days"

    local deleted_count=0
    local freed_space=0

    find "$LOG_DIR" -name "*.gz" -mtime +$MAX_AGE_DAYS -type f | while read -r old_log; do
        local file_size=$(stat -c%s "$old_log" 2>/dev/null || stat -f%z "$old_log" 2>/dev/nu

        if rm "$old_log"; then
            log "Deleted old log: $old_log ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((deleted_count++))
            freed_space=$((freed_space + file_size))
        fi
    done

    log "Cleanup completed: $deleted_count files deleted, $(numfmt --to=iec-i --suffix=B $fr
}

```

```

# Show log statistics
show_statistics() {
    echo "Log File Statistics"
    echo "======"
    echo "Generated: $(date)"
    echo

    # Count log files by type
    echo "Log Files by Extension:"
    echo "======"
    find "$LOG_DIR" -type f -name "*.log*" | sed 's/.*\./' | sort | uniq -c | sort -nr

    echo

    echo "Largest Log Files:"
    echo "======"
    find "$LOG_DIR" -type f -name "*.log*" -exec ls -lh {} \; | sort -k5 -hr | head -10

    echo

    echo "Disk Usage by Directory:"
    echo "======"
    du -h "$LOG_DIR"/* 2>/dev/null | sort -hr | head -10

    echo

    echo "Compressed vs Uncompressed:"
    echo "======"
    local compressed_count=$(find "$LOG_DIR" -name "*.gz" -type f | wc -l)
    local uncompressed_count=$(find "$LOG_DIR" -name "*.log" -type f | wc -l)
    local compressed_size=$(find "$LOG_DIR" -name "*.gz" -type f -exec stat -c%s {} \; 2>/dev/null | sort -nr | head -1 | cut -d' ' -f2)
    local uncompressed_size=$(find "$LOG_DIR" -name "*.log" -type f -exec stat -c%s {} \; 2>/dev/null | sort -nr | head -1 | cut -d' ' -f2)

    echo "Compressed files: $compressed_count $(numfmt --to=iec-i --suffix=B $compressed_size)"
    echo "Uncompressed files: $uncompressed_count $(numfmt --to=iec-i --suffix=B $uncompressed_size)"
}

# Main rotation process
main() {
    log "Starting log rotation process"

    if ! load_config; then
        log "Failed to load configuration, using defaults"
    fi

    # Get list of configured log sections
    local sections=()
    for key in "${!log_configs[@]}; do
        if [[ $key =~ ^([^.]+)\.files$ ]]; then

```

```

        sections+=("${BASH_REMATCH[1]}")
    fi
done

# Remove duplicates
IFS=$'\n' sections=($(sort -u <<<"${sections[*]}"))
unset IFS

if [ ${#sections[@]} -eq 0 ]; then
    log "No log sections configured, rotating all .log files in $LOG_DIR"

    # Rotate all .log files with default settings
    find "$LOG_DIR" -name "*.log" -type f | while read -r log_file; do
        rotate_log "$log_file" "default"
    done
else
    # Rotate configured logs
    for section in "${sections[@]"; do
        local files_pattern="${log_configs[$section.files]}"

        if [ -n "$files_pattern" ]; then
            log "Processing section: $section ($files_pattern)"

            # Expand file pattern
            for log_file in $files_pattern; do
                if [ -f "$log_file" ]; then
                    rotate_log "$log_file" "$section"
                fi
            done
        fi
    done
fi

# Cleanup old logs
cleanup_old_logs

log "Log rotation process completed"
}

# Show usage
show_usage() {
    cat << EOF
Log Rotator - Advanced log rotation utility

Usage: $0 [command] [options]

Commands:

```

rotate	Perform log rotation (default)
config	Generate configuration template
stats	Show log file statistics
cleanup	Clean up old compressed logs only

Options:

-c, --config FILE	Configuration file path
-d, --log-dir DIR	Log directory path
-a, --max-age DAYS	Maximum age for log cleanup
--help	Show this help

Environment Variables:

CONFIG_FILE	Configuration file path
LOG_DIR	Log directory path
MAX_AGE_DAYS	Maximum age for cleanup

Examples:

```
$0 # Rotate logs using default config
$0 config # Generate configuration template
$0 stats # Show log statistics
$0 cleanup --max-age 7 # Clean logs older than 7 days
```

EOF

}

Parse command line arguments

COMMAND="rotate"

while [[\$# -gt 0]]; do

case \$1 in

rotate|config|stats|cleanup)

COMMAND="\$1"

shift

;;

-c|--config)

CONFIG_FILE="\$2"

shift 2

;;

-d|--log-dir)

LOG_DIR="\$2"

shift 2

;;

-a|--max-age)

MAX_AGE_DAYS="\$2"

shift 2

;;

--help)

```

        show_usage
        exit 0
        ;;
    -*)
        echo "Unknown option: $1"
        show_usage
        exit 1
        ;;
    *)
        shift
        ;;
esac
done

# Execute command
case "$COMMAND" in
    rotate)
        main
        ;;
    config)
        generate_config
        ;;
    stats)
        show_statistics
        ;;
    cleanup)
        cleanup_old_logs
        ;;
    *)
        echo "Unknown command: $COMMAND"
        show_usage
        exit 1
        ;;
esac

```

These utility scripts provide comprehensive solutions for common file management and system maintenance tasks. Each script includes extensive configuration options, error handling, and detailed logging to make them suitable for production use.