Boring-NixOS

K19G

2026-09-12

Table of contents

Boring NixOS

Boring NixOS

A direct, practical guide to configuring infrastructure that builds identically every time: **deterministic, immutable, and refreshingly boring**.

Ambiguity is the enemy of reliability. When machines mutate silently through ad-hoc commands, builds drift, onboarding breaks, and outages follow. The boring default is to define the full closure in code — from developer compilers to multi-region cloud clusters — so every environment evaluates identically six months later.

Baseline: Nix **2.35+** · NixOS **26.05+** · Flakes enabled. Standalone and self-contained. You do not need any other title in this library.

Who this is for

- Engineers, platform developers, and SREs who want systems that build identically on a laptop, CI runner, and bare-metal server.
- Developers looking for instant, clean project environments without container lag or host library pollution.
- Teams moving away from fragile setup scripts and mutable system administration toward pure declarative configuration.

You do not need prior experience with Nix or functional programming. A terminal, a standard text editor, and basic Linux familiarity are all you need.

What "boring" means here

Nix is a purely functional package manager and domain-specific language. Applied with discipline, it eliminates entire categories of operational bugs: conflicting shared libraries, broken symlinks, broken OS upgrades, and undocumented host drift.

When over-complicated, it can become an impenetrable maze of nested overlays and tangled macros.

Boring NixOS teaches clear, sustainable defaults: - Plain, readable expressions over clever metaprogramming. - Explicit inputs pinned by cryptographic lockfiles. - Modular, auditable NixOS system definitions that scale without surprise.

How to read

1. Read the chapter concept and mental model.
2. Type the expression into its designated file (`default.nix`, `shell.nix`, `flake.nix`, or `configuration.nix`).
3. Run the exact build or evaluation command.
4. Experiment with the **Try this** exercises to verify how the store and evaluator behave.

The chapters are sequenced progressively. Install Nix (and optionally NixOS) in part 0, then the store and the language. You can jump to devShells, NixOS services, or CI caching once those basics exist.

How examples work

Every code listing is a complete, runnable expression or file. Nothing is an incomplete fragment left for you to guess.

```
# default.nix
let
  message = "reproducible desk online";
in
message
```

```
nix-instantiate --eval default.nix
```

```
"reproducible desk online"
```

A recurring scenario — a small team infrastructure desk (workstations, devShells, microservices, deployment pipelines) — connects the exercises so each technique solves a practical engineering challenge.

Map of the book

Part	Folder	What you leave with
0	`00-introduction`	Why imperative setups drift; install Nix; install NixOS on a disposable VM
1	`01-nix-fundamentals`	`/nix/store`, hashes, RUNPATH, closures, `db.sqlite`, GC, raw derivations, `why-depends` / `path-info` / `nix-tree`

Part	Folder	What you leave with
2	`02-nix-language`	Types, `let … in`, functions, attrsets, lists, imports, the REPL
3	`03-dev-environments`	`mkShell`, polyglot toolchains, flakes, direnv, nix-darwin, Nix on a foreign Linux, `nix-ld` / FHS
4	`04-packages-and-overlays`	nixpkgs, flake inputs, overlays, custom packages, fetchers, NIX_PATH / npins
5	`05-building-from-source`	`stdenv`, C/CMake, Rust, Python, Go modules / gomod2nix / Go shells / Go cross, Node, trivial builders, source filters, patches, anti-patterns
6	`06-nixos-system`	Modules, first machine, systemd, firewall, users, hardware, Disko, impermanence, generators, backups and GC windows, systemd Stage 1 initrd
7	`07-home-manager`	User modules, dotfiles, shell/editor, app settings, one-flake NixOS+HM
8	`08-cicd`	Binary caches, GitHub Actions, GitLab CI, caching strategy, tests, secrets, remote builders
9	`09-containers-k8s`	`dockerTools` images, k8s as modules, Helm, GitOps, mesh (optional late path)
10	`10-iac`	Terraform/OpenTofu providers, Ansible, cloud patterns
11	`11-secrets-security`	Why the store is public; sops-nix; Age; Vault; hardening
12	`12-platform-engineering`	IDP, monorepos, fleet, tuning, debug, tests, upgrades, Colmena, microVMs, CA internals
13	`13-production-capstone`	Threat model, Disko + ephemeral root, WireGuard, app stack, fleet rollback

Part	Folder	What you leave with
99	`99-appendices`	Glossary, command cheat sheet, Nix/NixOS version notes

What this book is not

- Not an exhaustive reference manual of all 100,000+ packages in `nixpkgs`.
- Not an exercise in esoteric functional programming puzzles.
- Not a guide to maintaining legacy channels and unpinned channels.

It is a concrete guide to writing reproducible systems that remain easy to understand, maintain, and operate.

Introduction

The Reproducibility Crisis

The Reproducibility Crisis

Software engineering has spent decades tolerating ambient state: hidden dependencies, global machine mutations, and "works on my machine" excuses. The boring default of this book is: **every development environment, dependency, and production system must be a pure, declarative, reproducible expression from zero**.

Mental model

Traditional operations rely on imperative mutation. You provision a virtual machine or container, run a sequence of `apt-get`, `curl | bash`, and `pip install` commands, and hope that: 1. Upstream package repositories haven't updated or removed files. 2. Network connections remain uninterrupted mid-install. 3. System state matches the exact sequence of commands previously tested.

When any of these fail, you get **Heisenbugs**: intermittent build failures, subtle ABI mismatches, and production outages that disappear when you attempt to isolate them.

```
Traditional Imperative Model:
  Base OS -> [Step 1: apt-get] -> [Step 2: pip install] -> [Step 3: git clone] -> ??? (Unknown

Nix Declarative Model:
  Inputs (Source + Hashes) -> Pure Derivation -> Immutable /nix/store Path (Deterministic)
```

Nix replaces stateful mutation with pure mathematical evaluation: given the exact same source code and inputs, it always yields the exact same byte-for-byte system closure.

Worked examples

Case 1: Detecting Ambient Host Contamination

Save as `test_environment.sh`. Inspect how standard shell scripts unknowingly depend on host binaries and paths.

```
# test_environment.sh
#!/usr/bin/env bash
set -euo pipefail

echo "Inspecting system path dependencies:"
which python3 || echo "python3 missing"
```

```
which node || echo "node missing"
which gcc || echo "gcc missing"
```

Run:

```
bash test_environment.sh
```

Output (on a typical developer workstation):

```
Inspecting system path dependencies:
/usr/bin/python3
/home/user/.nvm/versions/node/v20.0.0/bin/node
/usr/bin/gcc
```

Notice how three completely different tools live in three different unpinned locations (`/usr/bin`, user home directories, global OS paths). If a colleague runs the exact same script on a fresh laptop, it fails immediately.

Case 2: Purity and Isolated Evaluation

Save as `pure_check.nix`. A minimal Nix expression evaluated in isolation.

```
# pure_check.nix
let
  system = builtins.currentSystem;
  message = "No ambient host contamination allowed";
in
{
  inherit system message;
}
```

Run:

```
nix-instantiate --eval pure_check.nix
```

Output:

```
{ message = "No ambient host contamination allowed"; system = "x86_64-linux"; }
```

Evaluating this expression does not query the internet, look at `/usr/local`, or depend on environment variables.

Case 3: Hermetic Sandboxing in Action

Nix builds run in sandboxed namespaces where network access and non-declared filesystem paths are inaccessible.

Save as `sandbox_check.nix`:

```
# sandbox_check.nix
derivation {
  name = "sandbox-proof";
  system = builtins.currentSystem;
  builder = "/bin/sh";
  args = [
    "-c"
    "echo 'Built inside isolated namespace' > $out"
  ];
}
```

Run:

```
nix-build sandbox_check.nix
```

Output:

```
these 1 derivations will be built:
  /nix/store/10h9v...-sandbox-proof.drv
building '/nix/store/10h9v...-sandbox-proof.drv'...
/nix/store/v73ka...-sandbox-proof
```

Inspect output:

```
cat result
```

Output:

```
Built inside isolated namespace
```

The output was generated purely from the declared inputs.

The trap

Relying on “Golden Images” (pre-baked AMIs, Docker images without lockfiles, or manual server setup steps documented in internal wikis).

```
Wiki page: "Run these 14 commands to set up the desk service..."
```

By week three, one PPA repository goes offline, a patch release bumps a minor dependency, and onboarding is broken for every new hire.

The fix: Check in code that defines the environment completely down to cryptographic hashes. If it builds once, it builds forever.

The boring rule

- Never mutate system state in-place with ad-hoc shell commands.
- Pin all toolchain and library dependencies via cryptographic hashes.
- Eliminate dependency on the host's ambient `/usr/bin` and `$PATH`.
- Every environment must be reproducible from a fresh Git checkout.

Try this

1. Run `env` in your current shell and note how many variables point to user home directories or mutable local paths.
2. In `pure_check.nix`, add a field `timestamp = builtins.currentTime;` and observe how pure evaluation restricts non-deterministic built-ins.

Installing Nix

Installing Nix

The rest of this book assumes a working Nix installation with **flakes** and the **nix-command** CLI enabled. The boring default is: **a multi-user daemon install, flakes on, and `nix run nixpkgs#hello` succeeding before you write a single expression**.

You do not need NixOS yet. Nix the package manager runs on ordinary Linux and on macOS. NixOS (the operating system) comes in the next chapter.

Mental model

Nix is three things that share a name:

Piece	What it is
Nix the tool	The `nix` binary and the `nix-daemon` that builds and substitutes store paths
The store	`/nix/store`, a read-only tree of hashed paths
NixOS	A Linux distribution whose entire system is a Nix derivation

This chapter installs the first two. The daemon (multi-user) install is the one that matches production: unprivileged users talk to `nix-daemon` over a socket; builds run as dedicated `nixbld` users; the store is owned by root.

Single-user installs (`~/.nix-profile` only, no daemon) exist. Skip them on a shared workstation. They fight SELinux, they fight `/nix` permissions, and they do not match the NixOS mental model you will use later.

Flakes are still marked experimental in some docs. On this book's baseline (Nix 2.35+) they are the boring interface: `flake.nix` plus `flake.lock`, no ambient `<nixpkgs>` channel required.

```
curl installer


/nix created  +  nix-daemon enabled  +  your user in the trusted group


nix.conf: experimental-features = nix-command flakes


nix run nixpkgs#hello   →  "Hello, world!"
```

Worked examples

Case 1: Multi-user install on Linux

On a throwaway VM or a Linux workstation you control, run the official multi-user installer. Read the script. Then execute it:

```
curl -L https://nixos.org/nix/install -o nix-install.sh
less nix-install.sh
sh nix-install.sh --daemon
```

Output (abbreviated):

```
installing in multi-user mode...
alright! we're going to call sudo
copying Nix to /nix/store...
providing a `nix` command in your PATH...
setting up the nix-daemon systemd service...
Nix: just installed
```

Open a **new** login shell so `PATH` and `NIX_PROFILES` are set. Confirm:

```
nix --version
systemctl is-active nix-daemon
```

Output:

```
nix (Nix) 2.35.2
active
```

The daemon must be `active`. If it is not, `systemctl enable --now nix-daemon` and check `journalctl -u nix-daemon`.

Case 2: Enable flakes in nix.conf

The installer may already have turned flakes on. Check:

```
nix show-config | grep experimental-features
```

If the line does not include both `nix-command` and `flakes`, write them yourself.

On a multi-user install the file is `/etc/nix/nix.conf` (root). On a single-user leftover it is `~/.config/nix/nix.conf`. Prefer the system file:

```
# /etc/nix/nix.conf
experimental-features = nix-command flakes
```

Restart the daemon so the change is picked up:

```
sudo systemctl restart nix-daemon
```

`nix flake --help` must print help, not an "experimental feature disabled" error.

On a **multi-user** machine the daemon reads `/etc/nix/nix.conf`. Putting flakes only in `~/.config/nix/nix.conf` makes `nix flake` work in your shell and fail in `nix-daemon` builds (CI-shaped errors on the laptop). After editing the system file, `systemctl restart nix-daemon`.

26.05 NixOS ships daemon Nix **2.34**. This book's CLI baseline is **2.35+** (installer, or `nix.package` on NixOS). `nix --version` vs `nixos-option nix.package` — do not assume they match.

Case 3: First substitute from cache.nixos.org

You have not compiled anything. `nix run` should **download** a closure:

```
nix run nixpkgs#hello
```

Output:

```
copying path '/nix/store/…-hello-2.12.1' from 'https://cache.nixos.org'...
Hello, world!
```

If this compiles from source for minutes, your substituters are missing **or unsigned**. Confirm:

```
nix show-config | grep substituters
```

`https://cache.nixos.org` must be listed. The matching public key is shipped with Nix; you do not paste it by hand.

Case 4: A pinned flake instead of a floating channel

`nix run nixpkgs#hello` follows whatever `nixpkgs` your registry currently points at. That is fine for a smoke test. For desk work, pin.

Save as `flake.nix` in an empty directory:

```
# flake.nix
{
  description = "Desk smoke test";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
```

```
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      packages.${system}.default = pkgs.hello;
    };
}
```

If you are on aarch64, change `system` to `"aarch64-linux"` (or `"aarch64-darwin"` on Apple silicon).

```
nix flake lock
nix run
```

Output:

```
Hello, world!
```

`flake.lock` now pins the exact nixpkgs revision. Commit both files. The next machine that clones this directory gets the same `hello`.

Case 5: macOS (same daemon, different volume)

On macOS the installer creates an APFS volume for `/nix` (the root filesystem is signed and cannot hold `/nix` directly). Run the same official script with `--daemon`. After install:

```
nix --version
nix run nixpkgs#hello
```

Do not install Nix with Homebrew. Homebrew's `nix` package fights the daemon, the volume, and the store layout. One installer, one `/nix`.

The trap

The trap is **`curl | sh` without reading, then a second installer on top**. Mixing the official installer, the Determinate installer, and a distro package (`dnf install nix`, `apt install nix-bin`) produces two daemons, two `nix.conf` files, and a store the next `nix-collect-garbage` is afraid to touch.

Pick **one** installer and keep it:

Situation	Installer
Generic Linux VM, this book	Official multi-user (`--daemon`)
Team that already standardised on Determinate	Determinate, and only that
Fedora with a packaged `nix` you are required to use	The distro package, then enable flakes by hand

If you already have a broken mix:

```
# last resort: uninstall, then one clean install
sudo systemctl stop nix-daemon
# follow the uninstall steps for whichever installer you actually used
```

Do not `rm -rf /nix` while a daemon is running.

A second trap is enabling flakes in `~/.config/nix/nix.conf` while the daemon reads `/etc/nix/nix.conf`. Your shell says flakes are on; `nix-build` via the daemon says they are not. Put `experimental-features` in the file the **daemon** reads.

The boring rule

- Multi-user daemon install. One `/nix`. One `nix-daemon`.
- `experimental-features = nix-command flakes` in the daemon's `nix.conf`.
- Prove the install with `nix run nixpkgs#hello` substituting from `cache.nixos.org`, not compiling.
- Pin nixpkgs in a `flake.lock` as soon as you keep the directory.
- Do not stack installers. Do not install Nix via Homebrew.

Try this

1. Run `nix run nixpkgs#cowsay -- "desk online"` and confirm the NAR came from `cache.nixos.org` (the "copying path … from" line).
2. In Case 4's directory, run `nix flake metadata` and write down the locked nixpkgs revision. Change nothing, run it on a second machine (or a container with Nix), and confirm the revision matches.
3. Comment out `experimental-features` in `nix.conf`, restart the daemon, and run `nix flake show`. Restore the line after you have seen the error.
4. Run `ls -ld /nix /nix/store` and note the owners. The store should not be writable by your ordinary user.

Installing NixOS

Installing NixOS

Nix the package manager is not NixOS. NixOS is a Linux distribution whose kernel, initrd, systemd units, users, and `/etc` are one derivation. The boring default is: **install once on a disposable VM, generate hardware config, write a tiny `configuration.nix`, and treat `nixos-rebuild switch` as the only way the machine changes**.

Do not practise on the only disk that holds your photographs.

Mental model

The installer ISO is a live NixOS. You boot it, partition a disk, generate a hardware snapshot, write a system expression, and run `nixos-install` (plain, or `--flake .#host` once a flake exists). That produces generation 1. From then on you edit code and rebuild. You do not `apt install`. You do not `systemctl enable` by hand.

The 26.05 live ISO already has flakes. You do not need to `nix-env -iA nixos.git` just to clone the flake. `nixos-install --flake /mnt/etc/nixos#desk-vm` is the same install as the classic `/mnt/etc/nixos/configuration.nix` path, with a lockfile.

File	Role
`hardware-configuration.nix`	Disk UUIDs, file systems, kernel modules. Generated. Rarely edited.
`configuration.nix`	Hostname, users, packages, services, bootloader. You write this.
`/nix/var/nix/profiles/system`	Pointer at the current generation
Bootloader menu	Every previous generation you can roll back to

```
installer ISO

partition + mount  →  /mnt  and  /mnt/boot

nixos-generate-config --root /mnt

edit /mnt/etc/nixos/configuration.nix

nixos-install   →   reboot   →   generation 1
```

This book targets **NixOS 26.05+**. After install, `nixos-version` should agree.

Worked examples

Case 1: Disposable VM, not a daily driver

Create a UEFI VM:

Resource	Starting point
Firmware	UEFI
RAM	4 GB (8 GB is more comfortable)
Disk	40 GB
CPU	2 cores

Download the graphical or minimal NixOS 26.05 ISO from the NixOS download page. Attach it. Boot. You are in a live environment with a root shell (minimal ISO) or a desktop (graphical ISO).

Check firmware:

```
ls /sys/firmware/efi
```

If that directory exists, you are in UEFI mode. Match the VM firmware to the bootloader you will enable (`systemd-boot` for UEFI).

Case 2: Partition, mount, generate config

On a single empty virtual disk `/dev/vda` (names vary: `vda`, `sda`, `nvme0n1`):

```
lsblk
```

A boring GPT layout for UEFI:

Partition	Size	Type	Mount
`/dev/vda1`	1 GB	EFI System	`/mnt/boot`
`/dev/vda2`	rest	Linux filesystem	`/mnt`

Using `parted` and `mkfs` by hand is acceptable for this first machine. Disko (later in the book) replaces this for fleets. For one VM:

```
parted /dev/vda -- mklabel gpt
parted /dev/vda -- mkpart ESP fat32 1MiB 1GiB
parted /dev/vda -- set 1 esp on
parted /dev/vda -- mkpart primary ext4 1GiB 100%
```

```
mkfs.fat -F32 /dev/vda1
mkfs.ext4 /dev/vda2

mount /dev/vda2 /mnt
mkdir -p /mnt/boot
mount /dev/vda1 /mnt/boot
```

Generate the hardware snapshot:

```
nixos-generate-config --root /mnt
ls /mnt/etc/nixos/
```

Output:

```
configuration.nix
hardware-configuration.nix
```

Open `hardware-configuration.nix`. You should see the file-system UUIDs for `/` and `/boot`. Do not invent UUIDs. Leave this file as generated.

Case 3: A tiny first `configuration.nix`

Replace the generated `configuration.nix` with something you can read in one screen. Save as `/mnt/etc/nixos/configuration.nix`:

```
# configuration.nix
{ config, pkgs, ... }:

{
  imports = [ ./hardware-configuration.nix ];

  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;
  boot.loader.systemd-boot.configurationLimit = 8;

  networking.hostName = "desk-vm";
  time.timeZone = "UTC";
  i18n.defaultLocale = "en_US.UTF-8";

  users.users.deskadmin = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    # Lab only. Hashed at eval; the *hash* still lands in the store.
    # After first login: passwd, then delete this option.
    initialPassword = "changeme";
  };
```

```
  environment.systemPackages = with pkgs; [
    git
    vim
    curl
  ];

  services.openssh.enable = true;

  system.stateVersion = "26.05";
}
```

`stateVersion` is **not** "the release I want to be on." It is "the release whose defaults this machine was born with." Leave it at the release you installed. Changing it later without reading the release notes is how databases and state dirs break.

`configurationLimit = 8` keeps the ESP from filling with old kernels. The first-machine chapter adds GC; this is the bootloader side of the same hygiene.

Set a real password after first login (`passwd`). Then **remove** `initialPassword`. That option hashes at eval time; the hash is still world-readable in `/nix/store`. Production uses `hashedPasswordFile` (sops) or `users.mutableUsers = true` plus a one-shot `passwd`. `hashedPassword = "…"` in the module is the same store leak as `initialPassword`, just pre-hashed.

Install from the generated files (no flake yet):

```
nixos-install --no-root-passwd
nixos-enter --root /mnt -c 'passwd deskadmin'
reboot
```

`--no-root-passwd` skips the interactive root prompt when you already declared a wheel user. `nixos-enter` sets the password **inside** the installed root, not in the live ISO. `nixos-install` without that flag still works; it will ask for a root password.

Flake install (26.05 default once git exists). After `nixos-generate-config`, drop a flake next to the two Nix files. Save as `/mnt/etc/nixos/flake.nix`:

```
# flake.nix
{
  description = "Desk VM";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.desk-vm = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [ ./configuration.nix ];
    };
  };
}
```

```
nixos-install --flake /mnt/etc/nixos#desk-vm --no-root-passwd
```

The `#desk-vm` name is the `nixosConfigurations` attr, not a guess. `--root` defaults to `/mnt`. Copy `flake.lock` into git on first boot; the next rebuilds use `--flake .#desk-vm`. Do not `nixos-install` (no `--flake`) after you moved the machine to a flake — that builds the old channel `configuration.nix`.

Log in as `deskadmin`. Confirm:

```
nixos-version
hostname
```

Output:

```
26.05.9440.6aefcda (Yarara)
desk-vm
```

Case 4: The first rebuild on the installed system

You are now on the machine. Edit `/etc/nixos/configuration.nix` (or the flake in git). Add `ripgrep` to `environment.systemPackages` and activate. If you installed with `--flake`, keep using it:

```
sudo nixos-rebuild switch --flake /etc/nixos#desk-vm
```

Without `--flake`, this rebuilds the installer-era channel config. After git lives elsewhere, `--flake .#desk-vm` from the clone.

Output:

```
building the system configuration...
updating GRUB/systemd-boot...
activating the configuration...
setting up /etc...
reloading user units...
```

```
which rg
rg --version
```

`rg` is on `PATH`. It came from the new generation, not from a manual download.

List generations:

```
sudo nix-env --list-generations --profile /nix/var/nix/profiles/system
```

You should see generation 1 (install) and generation 2 (ripgrep). The bootloader menu lists both.

Case 5: Rollback without a rescue disk

Break the machine on purpose in the VM. In `configuration.nix`, set a nonsense timezone or remove `boot.loader.systemd-boot.enable` — something that still **builds** but is wrong — or, safer, add a package, switch, then:

```
sudo nixos-rebuild switch --rollback
```

Output:

```
switching to generation 1
activating the configuration...
```

`rg` is gone. The bootloader default is generation 1 again. If a rebuild leaves the system unbootable, pick the previous generation in systemd-boot (hold Space at firmware, then select the older NixOS entry).

The trap

The trap is **mutating the live system after install**. `passwd` for the first login is fine. `useradd`, `systemctl enable nginx`, `curl | sh` into `/usr` are not. Those changes vanish on the next rebuild, or worse, they survive in `/etc` as unmanaged files that fight the next activation.

A second trap is editing `hardware-configuration.nix` to "clean it up." The file is ugly because hardware is ugly. If you replace UUIDs with `/dev/vda2`, the next disk reorder will fail to boot.

A third trap is treating `stateVersion` as an upgrade knob. Upgrade by changing the flake input (or channel) to a new release and reading the release notes. Keep `stateVersion` at the original value unless a release note tells you to bump a specific option.

A fourth trap is shipping `initialPassword` (or `hashedPassword = "…"`) on a host that will ever be copied. `nix path-info -r /run/current-system` includes the hash. Rotate, then `hashedPasswordFile`.

The boring rule

- Practise on a VM. UEFI. One ESP, one root. `systemd-boot` + `configurationLimit`.
- `nixos-generate-config` owns `hardware-configuration.nix`. You own `configuration.nix` (and `flake.nix`).
- `nixos-install --flake /mnt/etc/nixos#host` once a flake exists. Same attr name as `nixosConfigurations`.
- `system.stateVersion` is the birth release. Leave it.
- After install, the only mutation path is `nixos-rebuild switch --flake` (or `test`, or `boot`).
- Know `--rollback` and the bootloader menu before you need them.
- `initialPassword` is lab-only. Delete it after `passwd`.

Try this

1. After Case 4, run `sudo nixos-rebuild test` with a hostname change. Confirm `hostname` changed, reboot, and confirm it reverted (test does not update the bootloader default).
2. Run `sudo nixos-rebuild dry-activate` and read which units would restart. No change should be applied.
3. Add `cowsay` to `environment.systemPackages`, switch, run `cowsay desk`, then rollback and confirm `cowsay` is missing.
4. Open `hardware-configuration.nix` and write down the UUID of `/`. Compare with `lsblk -f`. They must match.
5. Install a second VM with `--flake /mnt/etc/nixos#desk-vm`. Confirm `nixos-version` still says 26.05 and `ls /etc/nixos/flake.lock` exists.

Nix Fundamentals

Store Model

The /nix/store Anatomy

The /nix/store Anatomy

Everything Nix builds, downloads, or installs lands in `/nix/store`. The boring default is: **treat the store as a read-only database of paths named by hash, and never move, rename, or chmod anything inside it**.

Mental model

Every store path looks like this:

`/nix/store/<hash>-<name>-<version>`

Example:

`/nix/store/z1k947f6y788sfaivs931h26hsvb3506-ripgrep-14.1.0`

Piece	Meaning
Hash	32 characters of Nix base32. Identity of this path.
Name + version	Human label. Not identity. Two `ripgrep-14.1.0` builds can coexist.
The directory	A complete artifact: `bin/`, `lib/`, `share/`, or a single file.

The hash is computed from the **inputs** of the derivation (the usual case) or from the **contents** of the output (fixed-output and content-addressed paths). Either way, a different hash is a different path. Nothing is overwritten.

Permissions are read-only. Timestamps are the Unix epoch (`Jan 1 1970`) so two machines producing the same bytes do not disagree about mtime.

```
/nix
   store/          immutable artifacts
   var/nix/
        db/        SQLite index of those artifacts
        profiles/  generations (GC roots)
        gcroots/   extra roots (result, direnv, …)
```

Your shell's `rg` is not "the ripgrep in /usr/bin". It is a symlink into a specific store path.

Worked examples

Case 1: Follow a binary into the store

```
which rg
ls -l $(which rg)
ls -ld $(dirname $(which rg))/..
```

Output (hashes will differ on your machine):

```
/run/current-system/sw/bin/rg
lrwxrwxrwx 1 root root 68 Jan  1  1970 /run/current-system/sw/bin/rg -> /nix/store/z1k947f6y788
dr-xr-xr-x 1 root root 4096 Jan  1  1970 /nix/store/z1k947f6y788sfaivs931h26hsvb3506-ripgrep-14
```

On a Nix-on-Linux user profile the first hop may be `~/.nix-profile/bin/rg` instead of `/run/current-system`. The last hop is always `/nix/store/…`.

Case 2: Read-only and epoch timestamps

```
ls -ld $(readlink -f $(which nix))
```

Output:

```
-r-xr-xr-x 1 root root 15728640 Jan  1  1970 /nix/store/7vqw9388aaaaaaaaaaaaaaaaaaaaaaaa-nix-2.
```

- Mode `r-xr-xr-x` (or `r-xr-xr-x` on the directory `r-xr-xr-x`): no write bit.
- Date `Jan 1 1970`: not "this was built at the epoch"; the timestamp is **normalised** so it is not an input to the next hash.

Try to touch it:

```
touch $(readlink -f $(which nix))
```

Output:

```
touch: cannot touch '/nix/store/…-nix-2.35.2/bin/nix': Permission denied
```

Case 3: Two versions, no collision

```
nix build nixpkgs#hello --out-link hello-a
ls -ld hello-a
```

Output:

```
lrwxrwxrwx 1 deskadmin deskadmin 55 Jan  1 12:00 hello-a -> /nix/store/…-hello-2.12.1
```

`hello-a` is a **GC root** in the current directory (the `result`-style symlink). The store path is the real artifact. Installing a different nixpkgs revision of `hello` produces a second directory next to the first. Both stay until nothing roots them.

Case 4: What is *not* in the store

```
ls /usr/bin/rg 2>&1 || true
echo $PATH | tr ':' '\n' | head
```

On NixOS, `/usr/bin` is nearly empty (`env`, maybe `sh`). Tools live under `/run/current-system/sw/bin` and `/nix/store`. On Nix-on-Linux, `/usr/bin` still has the host distro; Nix tools are extra entries on `PATH`. Mixing them is how "it works in my shell" happens. Prefer the store path.

Case 5: Name a path without guessing the hash

```
nix path-info nixpkgs#hello
```

Output:

```
/nix/store/…-hello-2.12.1
```

`nix path-info -Sh nixpkgs#hello` adds closure size. You do not construct store paths by concatenating strings in a shell script.

The trap

The trap is **`sudo chmod -R u+w /nix/store/…` so you can patch a file**. The path's hash promised those bytes. After you write to it, every later build that trusted that path is lying. The daemon may also refuse to register the path on `--verify`.

If a binary is wrong, rebuild it with different inputs. You get a new hash. The old path stays until GC.

A related trap: copying a store path to another machine with `scp` and dropping it under `/nix/store` by hand. Use `nix copy` so the database is updated. A directory that is not registered in `db.sqlite` is invisible to Nix and eligible for nothing useful.

The boring rule

- All Nix artifacts live under `/nix/store/<hash>-<name>`.
- Follow `which` until you see `/nix/store`. That is the real path.
- Never `rm`, `mv`, `chmod`, or `touch` inside the store.
- Use `nix path-info` / `nix-store -q` to name paths. Do not paste hashes from memory.
- Treat `result` and profile symlinks as pointers, not as the software.

Try this

1. Run `readlink -f $(which bash)` (or `$(which sh)`) and count how many symlink hops precede `/nix/store`.
2. `ls /nix/store | wc -l` — that is how many artifacts this machine currently holds. Run it again after `nix run nixpkgs#cowsay -- hi` and see the count move.
3. `stat $(readlink -f $(which nix))` and confirm `Access: (0555)` (or similar, no write) and a 1970 timestamp.
4. Try `echo x >> $(readlink -f $(which nix))` and save the exact error. That error is the product.

Content-Addressable Paths

Content-Addressable Paths

Most Nix store paths you meet are **input-addressed**: the hash is a digest of the derivation (builder, env, input paths), not of the bytes that came out. **Content-addressed** (CA) paths hash the output itself. The boring default is: **understand the difference, keep producing ordinary input-addressed builds, and let fixed-output derivations (fetchers) be the CA paths you actually use every day**.

Mental model

Kind	Hash comes from	Same output, different builder?
Input-addressed	The `.drv` (inputs, builder, env)	Two different store paths
Content-addressed / FOD	The output bytes (and hash mode)	Same store path

Input-addressed is why a comment in a build script rebuilds everything downstream: the `.drv` changed, so the output path changed, even if `gcc` emitted identical bytes.

Fixed-output derivations (the `fetchurl` / `fetchFromGitHub` family) are the everyday CA case. You declare `hash = "sha256-…"`; Nix may use the network; it then **checks** the bytes against that hash. The store path is a function of the hash, not of the URL. Change the URL to a mirror, keep the hash, keep the path.

Experimental CA derivations for *ordinary* builds exist (`contentAddressed` in Nix). They are not the boring default for a desk workstation. The platform-engineering internals chapter covers them later.

```
Input-addressed:
  .drv   hash    /nix/store/AAAA-desk-api
                     (even if bytes == yesterday's BBBB-desk-api)

Fixed-output:
  declared hash      /nix/store/HHHH-source.tar.gz
  (URL is just how we get the bytes; the hash is identity)
```

Worked examples

Case 1: Input hash changes when the builder changes

Save as `banner-a.nix`:

```
# banner-a.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-banner";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [ "-c" "echo 'Desk System Online' > $out" ];
}
```

Save as `banner-b.nix` — same bytes out, extra comment in the script:

```
# banner-b.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-banner";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [ "-c" "echo 'Desk System Online' > $out  # lab note" ];
}
```

```
nix-instantiate banner-a.nix
nix-instantiate banner-b.nix
```

Output:

```
/nix/store/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa-desk-banner.drv
/nix/store/bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb-desk-banner.drv
```

Two `.drv` files. Two future output paths. The echo is the same; the input is not.

Case 2: Store paths are immutable, not “updated in place”

```
touch $(readlink -f $(which nix)) 2>&1 || true
```

Output:

```
touch: cannot touch '/nix/store/…-nix-2.35.2/bin/nix': Permission denied
```

Nix never overwrites `AAAA-desk-banner` with new bytes. It builds `CCCC-desk-banner` and swings a profile symlink. That is input-addressing plus immutability working together.

Case 3: Fixed-output identity is the hash

Save as `fod.nix`:

```
# fod.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.fetchurl {
  url = "https://cache.nixos.org/nix-cache-info";
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build fod.nix
```

The build **fails** (unless you lucked into that hash). The error is the point:

```
error: hash mismatch in fixed-output derivation '/nix/store/…-nix-cache-info':
         specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
            got:    sha256-<the real hash>
```

Paste the `got` value into `hash`, rebuild, and the path is now determined by those bytes. Point `url` at a different mirror of the **same** file and the store path stays put.

Case 4: Why CA helps a cache

If two CI jobs fetch the same tarball with the same hash, they share one store path. If two CI jobs compile the same C file with two different comment-only builder changes, input-addressed Nix stores two outputs even when `cmp` would say the binaries match.

That waste is real. It is also cheaper than debugging "why did the binary change when the source didn't" because a timestamp leaked into an input. Boring teams live with input-addressed builds and pin fetch hashes.

Case 5: Inspect which kind a path is

```
nix path-info --json $(nix-build '<nixpkgs>' -A hello --no-out-link) | head
```

Look for `ca` / `narHash` fields when present. Ordinary `hello` from nixpkgs is input-addressed. A `fetchurl` output is fixed-output. You do not need to memorise JSON; you need to know which family you are looking at when a cache misses.

The trap

The trap is **turning off the hash** (`hash = "";` or `sha256 = lib.fakeSha256` left in production) so the fetch "just works." The next time GitHub retags, or a mirror is replaced, you silently take new bytes at the same URL. The whole point of a FOD is the mismatch error.

The other trap is enabling experimental CA derivations on a laptop because a blog post promised automatic sharing. Keep CA-for-builds in the lab until the whole team agrees. Fetchers already gave you the CA that matters.

The boring rule

- Everyday builds are input-addressed. A `.drv` change is a new path, even if the bytes look the same.
- Fetchers are fixed-output. Identity is the **hash you wrote**.
- Never leave a dummy hash in a committed file.
- Do not chmod the store to "fix" a path. Build a new one.
- Treat experimental content-addressed builds as optional, not as the desk default.

Try this

1. Change only the `name` in `banner-a.nix` from `desk-banner` to `desk-banner-v2`, instantiate, and confirm the `.drv` hash moved.
2. Run Case 3, copy the `got` hash, rebuild, then run `nix path-info -Sh result` and note the size.
3. Flip one character of the working hash and read the mismatch error. That error is success: Nix refused the bytes.
4. Explain in one sentence, to a colleague, why `hello` from nixpkgs and a `fetchurl` tarball are hashed differently.

Hashes and Dependencies

Cryptographic Hashes and Inputs

Cryptographic Hashes and Inputs

Two machines produce the same store path only if they evaluate the same derivation. The boring default is: **every input — source, compiler, flags, patches, build script — is part of the hash, and a one-bit change is a new path, not an in-place upgrade**.

Mental model

Traditional package managers treat `ripgrep-14.1.0` as identity. It is a label. A `rg` built against glibc 2.38 is not the same program as a `rg` built against glibc 2.39.

Nix hashes the `.drv` file: a Nix-language serialisation of the builder, environment, and input store paths. The output path is `/nix/store/<hash-of-that-drv>-<name>`.

```
sources + toolchain + env + builder


         .drv file    SHA    output path


      sandbox build      bytes at that path
```

If the build is allowed to see the network, the time of day, or `/usr/bin/cc`, the hash is a lie. The sandbox exists so the hash means something.

Worked examples

Case 1: Instantiate before you build

Save as `sample.nix`:

```nix
# sample.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-sample";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
```

```
  args = [ "-c" "echo ok > $out" ];
}
```

```
nix-instantiate sample.nix
```

Output:

```
/nix/store/7vqw…-desk-sample.drv
```

No compiler ran. The hash is already decided. `cat` that `.drv` (it is text) and you will see the builder path, `name`, and `system`.

Case 2: One flag, new path

Save as `opt.nix`:

```
# opt.nix
{ cflags ? "-O2" }:
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-opt";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  inherit cflags;
  args = [ "-c" "echo $cflags > $out" ];
}
```

```
nix-instantiate opt.nix
nix-instantiate --argstr cflags -O3 opt.nix
```

Output:

```
/nix/store/aaaa…-desk-opt.drv
/nix/store/bbbb…-desk-opt.drv
```

`-O2` versus `-O3` is two derivations. Both can live on disk. Nothing was upgraded in place.

Case 3: The compiler is an input

```
nix-instantiate '<nixpkgs>' -A hello
nix-store -q --references $(nix-instantiate '<nixpkgs>' -A hello)
```

Output (abbreviated):

```
/nix/store/…-hello-2.12.1.drv
/nix/store/…-stdenv-linux.drv
/nix/store/…-source
…
```

`hello` does not hash "a C compiler." It hashes **this** stdenv, which hashes **this** gcc, which hashes **this** source tarball. That is why a nixpkgs bump rebuilds the world: gcc moved, so everything that listed gcc as an input moved.

Case 4: Purity — the sandbox cannot see `/usr`

Save as `impure.nix`:

```
# impure.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-impure";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [ "-c" "/usr/bin/id > $out" ];
}
```

```
nix-build impure.nix
```

Output:

```
building '/nix/store/…-desk-impure.drv'...
/bin/bash: line 1: /usr/bin/id: No such file or directory
error: builder for '/nix/store/…-desk-impure.drv' failed
```

The sandbox does not contain `/usr`. That failure is the hash doing its job: if `/usr/bin/id` had been used, two machines with different `id` binaries would still have produced the "same" store path.

Case 5: Pin the input set with a lockfile

```
nix flake new ./desk-pin
cd desk-pin
nix flake lock
nix hash path flake.lock
```

`flake.lock` records the git revision and NAR hash of `nixpkgs`. Two laptops with the same lock file instantiate the same `.drv` hashes. Without the lock, `nixpkgs` is a moving label.

The trap

The trap is `nix-build --impure` (or `builtins.currentTime`, or `filterSource` that includes `.git`) so a "quick test" leaks host state into the hash. The path then differs between CI and a laptop for reasons that do not show up in `git diff`.

A second trap is trusting version numbers in `pname` / `version` as identity. They are labels inside the path. The hash is identity. Rename freely; never assume `1.0.0` means the same bytes as last week's `1.0.0`.

The boring rule

- The `.drv` is hashed **before** the build. Instantiate to see identity.
- Toolchains, flags, and sources are inputs. Changing any of them is a new path.
- Keep the sandbox on. A build that needs `/usr` is not a Nix build.
- Pin nixpkgs (flake lock or a pinned fetch). Do not build against a floating channel on a desk that ships artifacts.
- Version strings are labels. Hashes are names.

Try this

1. Add an environment variable `desk = "window-a";` to `sample.nix`, instantiate, and confirm the `.drv` hash changed even though `echo ok` did not.
2. Run `nix-store --query --binding builder $(nix-instantiate sample.nix)` and confirm it points at bash in `/nix/store`, not `/bin/bash`.
3. Build `opt.nix` twice (`-O2` and `-O3`), then `diff -u $(nix-build opt.nix) $(nix-build --argstr cflags -O3 opt.nix)`.
4. Search a `.drv` with `grep gcc $(nix-instantiate '<nixpkgs>' -A hello)` and note that gcc appears as a store path, not as the word "gcc".

Deterministic Dynamic Linking and RPATH

Deterministic Dynamic Linking and RPATH

On a conventional Linux system, `rg` asks the dynamic linker for `libc.so.6` and takes whatever `/lib` currently holds. A glibc upgrade then changes every running binary at once. The boring default is: **bake the exact store path of every needed library into the ELF `RUNPATH`, and leave `LD_LIBRARY_PATH` alone**.

Mental model

When Nix links a binary it sets `DT_RUNPATH` (or older `DT_RPATH`) to store directories:

```
ELF header
  RUNPATH = /nix/store/7r44p…-glibc-2.39/lib:/nix/store/…-gcc-…/lib
```

`ld.so` looks **there**, not in `/lib`, not in `/usr/lib`. Two `rg` binaries can sit on the same machine, each with its own glibc, and neither notices the other.

Reference scanning then records those store paths as **runtime references**. Garbage collection will not delete glibc while a rooted `rg` still points at it.

```
rg  --RUNPATH-->  glibc-2.39/lib
      --references (db.sqlite) -->  same glibc path
```

The program interpreter (`ld-linux`) is also a store path. A Nix binary copied to a machine without `/nix` dies with `No such file or directory` even though the file is there.

Worked examples

Case 1: Read RUNPATH off a real binary

```
rg_bin=$(readlink -f $(which rg) 2>/dev/null || echo "$(nix path-info nixpkgs#ripgrep)/bin/r
readelf -d "$rg_bin" | grep -E 'RPATH|RUNPATH'
```

Output (shape):

```
0x000000000000001d (RUNPATH)            Library runpath: [/nix/store/7r44p12l4b72j9m9l09k96ygh
```

If `rg` is a wrapper script, follow it with `readlink -f` or inspect the path under `nix path-info`.

Case 2: Which libraries would actually load

```
ldd "$rg_bin"
```

Output (abbreviated):

```
linux-vdso.so.1 (0x00007ffd…)
libgcc_s.so.1 => /nix/store/…-xgcc-…-libgcc/lib/libgcc_s.so.1
libc.so.6 => /nix/store/…-glibc-2.39/lib/libc.so.6
/nix/store/…-glibc-2.39/lib/ld-linux-x86-64.so.2
```

Every `=>` target is under `/nix/store`. Nothing resolves to `/lib/x86_64-linux-gnu`.

Case 3: The interpreter is a store path too

```
readelf -l "$rg_bin" | grep interpreter
```

Output:

```
[Requesting program interpreter: /nix/store/…-glibc-2.39/lib/ld-linux-x86-64.so.2]
```

The kernel loads **that** `ld-linux`, not `/lib64/ld-linux-x86-64.so.2`. `scp bin/rg` to a host without `/nix` fails here, not because the bytes of `rg` are missing.

Case 4: Runtime closure matches RUNPATH

```
nix path-info -r $(nix path-info nixpkgs#ripgrep)
```

You should see glibc (and a few others), not gcc, not the source tarball. Compilers are build-time; `RUNPATH` is runtime. Confirm the same hash:

```
readelf -d "$rg_bin" | grep RUNPATH
nix path-info -r nixpkgs#ripgrep | grep glibc
```

Same `7r44p…` (or whatever 26.05 pinned).

Case 5: `LD_LIBRARY_PATH` punches a hole

```
LD_LIBRARY_PATH=/usr/lib ldd "$rg_bin" | head
```

Depending on the host, `ldd` may now show `/usr/lib` for some libraries. That is a **different program** than the one Nix hashed. It may work today and segfault after a distro upgrade.

Nix wrappers sometimes set `LD_LIBRARY_PATH` for a specific package that was not patched. That wrapper is part of the derivation. Your shell profile setting a global `LD_LIBRARY_PATH` is not.

Vendor ELF files get `nix-ld` or `buildFHSEnv`, not `export LD_LIBRARY_PATH` in `.bashrc`.

To see Nix *set* RUNPATH, build a tiny C program:

```
# rpath-demo.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.stdenv.mkDerivation {
  pname = "desk-rpath";
  version = "1.0";
  src = pkgs.writeTextDir "main.c" ''
    int main(void) { return 0; }
  '';
  buildPhase = ''
    $CC $src/main.c -o desk-rpath
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp desk-rpath $out/bin/
  '';
}
```

```
nix-build rpath-demo.nix
readelf -d result/bin/desk-rpath | grep -E 'RPATH|RUNPATH'
```

`$CC` (not `gcc`) is how stdenv injects the store RUNPATH. Hardcoding `gcc` from the host skips that. `NIX_LDFLAGS` / the bintools wrapper is what actually writes `DT_RUNPATH`. `dontPatchELF = true` skips Nix's patchelf — only when you know the binary is already right (or static).

Vendor blobs you **did** copy into `$out`:

```
# patchelf.nix fragment
{
  nativeBuildInputs = [ autoPatchelfHook ];
  buildInputs = [ openssl zlib ];
}
```

`autoPatchelfHook` walks `$out`, sets interpreter + RUNPATH from `buildInputs`. That is how a downloaded `desk-toolbox` becomes a Nix package. It is not `export LD_LIBRARY_PATH`.

The trap

The trap is **exporting `LD_LIBRARY_PATH` in `.bashrc` to make a vendor binary start**. Every Nix binary in that shell now prefers the vendor's `libssl.so`. The next OpenSSL update on the host becomes a Nix outage.

A second trap is `patchelf --set-rpath /usr/lib` on a store path. The path is read-only, and even if you copy it out, you have left the closure Nix knows about.

A third: calling host `gcc` in `buildPhase` so RUNPATH is empty and `ldd` shows `/lib`. A fourth: `dontPatchELF = true` copied from a gist so "the build is faster," then `libc.so.6 => not found`.

The boring rule

- Trust `RUNPATH`. Inspect it with `readelf -d` and `ldd`. Use `$CC`, not host `gcc`.
- Runtime references in the Nix database should match those libraries.
- Do not set global `LD_LIBRARY_PATH`, `LD_PRELOAD`, or `DYLD_*` to "fix" Nix.
- A binary copied off the store needs its interpreter and `RUNPATH` closure (`nix copy`, not `scp bin/rg`).
- Vendor ELF files get `nix-ld`, `buildFHSEnv`, or `autoPatchelfHook` — not a profile hack.

Try this

1. Compare `readelf -d $(which curl)` on a Nix `curl` versus, if you have one, `/usr/bin/curl`. Note where `libc` lives in each.
2. Run `nix path-info -Shr nixpkgs#ripgrep` and find glibc in the closure. Confirm the same hash appears in `RUNPATH`.
3. `file $(readlink -f $(which rg))` — confirm it is dynamically linked. (A few Nix packages are static; they have no `RUNPATH`. That is also fine.)
4. In a subshell, `export LD_LIBRARY_PATH=/does/not/exist` and run `rg --version`. Then unset it. If something fails only with the export, you have felt the hole.

Dependency Graphs

Dependency Closures and Graphs

Dependency Closures and Graphs

A store path is never alone. The boring default is: **know the runtime closure (what you must copy to production) versus the build-time closure (what the sandbox needed), and measure both with `nix-store` / `nix path-info` before you ship**.

Mental model

Nix records **direct references** by scanning output files for the 32-character hash of other store paths. If `rg` contains the string `/nix/store/7r44p…-glibc-2.39`, glibc is a runtime reference. `gcc` was used to compile `rg` but does not appear in the binary, so it is **not** in the runtime closure.

Term	Command	What you get
Direct references	`nix-store -q --references P`	Immediate edges out of `P`
Referrers	`nix-store -q --referrers P`	Paths that point **at** `P`
Runtime closure	`nix-store -q --requisites P`	`P` plus everything it needs to run
Deriver	`nix-store -q --deriver P`	The `.drv` that built `P`

```
build-time:  gcc, headers, cmake, fetchers
                  (sandbox only)

output:      rg   references    glibc, libgcc, …


runtime closure = rg + those references, transitively
```

Deploy the runtime closure. Do not deploy gcc because you compiled on this machine. `result` is a symlink to **one** path; the closure is that path plus requisites.

Worked examples

Case 1: Runtime closure of ripgrep

```
nix-store -q --requisites $(nix path-info nixpkgs#ripgrep)
```

Output (typical shape on 26.05):

```
/nix/store/7r44p12l4b72j9m9l09k96yghz02f8n7-glibc-2.39
/nix/store/…-libgcc-…
/nix/store/z1k947f6y788sfaivs931h26hsvb3506-ripgrep-14.1.1
```

A handful of paths. Not a compiler toolchain.

Case 2: Sizes, not just names

```
nix path-info -Shr nixpkgs#ripgrep
```

Flag	Meaning
`-S`	Size of this path
`-h`	Human units
`-r`	Closure (requisites)

The last line is the closure you would copy with `nix copy`. If that number is hundreds of megabytes for a "small" CLI, a wrapper script has pulled in Python or glib.

```
/nix/store/…-ripgrep-…      5.2M
…
narSize (closure)           32M
```

Numbers move with the 26.05 pin; the shape does not.

Case 3: Direct references versus the whole tree

```
echo 'references:'
nix-store -q --references $(nix path-info nixpkgs#ripgrep)
echo 'requisites:'
nix-store -q --requisites $(nix path-info nixpkgs#ripgrep)
```

References are one hop. Requisites are the transitive closure. For `rg` they are often almost the same. For a desktop environment they are not — `nix path-info -Shr /run/current-system` is the scary number.

Case 4: Why gcc is missing from runtime

```
out=$(nix path-info nixpkgs#hello)
drv=$(nix-store -q --deriver "$out")
echo "output: $out"
echo "deriver: $drv"
nix-store -q --references "$drv" | grep -E 'gcc|stdenv' | head
nix-store -q --references "$out"
```

The `.drv` references stdenv (gcc, binutils, …). The **output** of `hello` does not. Reference scanning dropped the toolchain. That is the entire point of separate build-time and runtime graphs.

Case 5: Copy what production needs

```
nix copy --to file://$PWD/desk-nar nixpkgs#ripgrep
ls desk-nar/nar | wc -l
nix path-info --store file://$PWD/desk-nar -r nixpkgs#ripgrep | grep gcc || echo 'no gcc in
```

The NAR cache contains the runtime closure, not gcc. A colleague can `nix copy --from file://$PWD/desk-nar` and run `rg` without compiling.

On a live cache this is `nix copy --to ssh://desk-cache` or a Cachix push. Same graph.

If the runtime closure of `desk-api` contains `gcc` or `python3` you did not mean to ship:

```
nix why-depends $(nix path-info nixpkgs#ripgrep) nixpkgs#gcc || echo 'gcc not in runtime (go
```

Cut the edge (wrapper, `propagatedBuildInputs`, shebang) rather than deleting gcc from the store by hand.

```
nix path-info -Sh nixpkgs#hello
nix path-info -Shr nixpkgs#hello
```

The first number is one path. The second is the closure you would `nix copy`. Write both down before you call a package “small.”

Interactive tree (optional extra):

```
nix run nixpkgs#nix-tree -- $(nix path-info nixpkgs#ripgrep)
```

Arrow keys walk references. `nix-tree` is a viewer, not a substitute for `path-info -Shr` in scripts.

The trap

The trap is **shipping `result` by scp** — one directory — and discovering the binary dies on the server because glibc is not there. `result` is a symlink to **one** path. The closure is that path plus requisites.

The other trap is reading `--requisites` on a `.drv` and panicking at the size. A `.drv`'s requisites include build-time inputs. Query the **output** path (`nix path-info nixpkgs#ripgrep`), not the deriver, when you mean "what runs."

The boring rule

- Runtime closure = `--requisites` of the **output** path.
- Measure with `nix path-info -Shr` before you call a package "small."
- Deploy with `nix copy` (or a binary cache), never with a single `scp` of `bin/`.
- Build-time tools belong in the `.drv`, not in production.
- If a runtime closure contains a compiler, a wrapper or `propagatedBuildInputs` leaked. Fix the package.

Try this

1. Compare `nix path-info -Sh nixpkgs#hello` (one path) with `nix path-info -Shr nixpkgs#hello` (closure). Write down both numbers.
2. Run `nix-store -q --graph $(nix path-info nixpkgs#hello)` and pipe to `head`. You should see `hello` and `glibc` nodes.
3. `nix-store -q --referrers $(nix path-info nixpkgs#glibc | head -1)` — many packages point at glibc. That is why it survives GC.
4. Intentionally `scp` only the `hello` store directory to a tmp dir and try to run it. Then explain the interpreter error with `readelf -l` (the RPATH chapter).

The Nix Database (db.sqlite)

The Nix Database (db.sqlite)

`/nix/store` is a directory tree. Nix does not discover it by walking the tree on every command. The boring default is: **the daemon owns `/nix/var/nix/db/db.sqlite`, and you query it with `nix-store` / `nix path-info`, never with `sqlite3`.**

Mental model

The database answers three questions:

1. Is this path **valid** (registered, complete, not half-copied)?
2. What does it **reference**, and what **refers to** it?
3. Which `.drv` **derived** it?

```
/nix/store/          bytes
/nix/var/nix/db/     index of those bytes
     db.sqlite
     schema
```

If you copy files into `/nix/store` with `cp`, the database does not know them. `nix-store --query` will not list them. GC will not preserve them. The next `--verify` may delete them as garbage.

The daemon holds a lock on the database. That is why two `nix-build` processes can run at once without corrupting registration. On a multi-user 2.35 install your account talks to the daemon; the daemon talks to SQLite.

Worked examples

Case 1: Referrers — who keeps this path alive?

```
nix-store -q --referrers $(nix path-info nixpkgs#ripgrep)
```

Output (shape):

```
/nix/store/z1k947f6y788sfaivs931h26hsvb3506-ripgrep-14.1.1
/nix/store/…-system-path
```

A path with **no** referrers and **no** GC root is collectable. Profiles and `result` symlinks show up as referrers once they point here.

```
nix-store -q --referrers-closure $(nix path-info nixpkgs#hello) | wc -l
```

That number is "how much of the graph would notice if hello vanished," not disk size.

Case 2: Valid path check

```
nix-store --query --hash $(nix path-info nixpkgs#hello)
nix-store --verify --check-contents 2>&1 | tail
```

`--verify --check-contents` rehashes every registered path. It is slow; it is the tool you run after a disk scare, not every morning. Without `--check-contents` it only checks that the directory exists.

Output on a healthy store:

```
reading the Nix database...
checking path existence...
checking hashes...
```

No "path is corrupted" lines.

Case 3: Deriver link

```
nix-store -q --deriver $(nix path-info nixpkgs#hello)
nix path-info --json nixpkgs#hello | jq '.[].deriver, .[].narHash, .[].references'
```

Output (shape):

```
/nix/store/…-hello-2.12.1.drv
```

Substituted paths (downloaded from a cache, never built here) may report `unknown-deriver`. That is not corruption. The bytes arrived without the `.drv`.

Case 4: Registration is why `nix copy` exists

```
mkdir -p /tmp/not-a-store
cp -a $(nix path-info nixpkgs#hello) /tmp/not-a-store/
nix-store -q --requisites /tmp/not-a-store/hello-* 2>&1 || true
```

Output:

```
error: path '/tmp/not-a-store/…' is not valid
```

The bytes exist. Nix does not care. Register them:

```
nix copy --to file:///tmp/desk-cache nixpkgs#hello
nix path-info --store file:///tmp/desk-cache nixpkgs#hello
```

That cache has its own database. USB `cp` of a store directory does not.

Case 5: Where the file lives (look, don't poke)

```
sudo ls -l /nix/var/nix/db/
```

Output:

```
-rw-r--r-- 1 root root … db.sqlite
-rw-r--r-- 1 root root … schema
```

Owned by root. You do not `sqlite3` this file. After filesystem damage:

```
sudo nix-store --verify --check-contents
```

Then restore missing paths from the team cache (`nix copy --from …`). Do not INSERT rows by hand.

The trap

The trap is **opening `db.sqlite` in a GUI and deleting a row** to "free space." The store directory and the database then disagree. `--verify` will complain; GC may delete the wrong thing; the daemon may refuse to start.

The other trap is `sudo rm -rf /nix/store/some-hash-…` because `du` scared you. The database still lists the path as valid. The next build that expected those bytes fails in a way that looks like a nixpkgs bug.

Use `nix-collect-garbage`. Use `nix-store --delete` only on a path with zero referrers, and only when you mean it.

The boring rule

- Query with `nix-store -q` and `nix path-info`. Never write SQL against `db.sqlite`.
- Paths that are not registered are not Nix paths, even if they sit under `/nix/store`.
- After filesystem damage, `nix-store --verify --check-contents`. Then restore from a cache, do not patch the DB.
- Substituted artifacts may lack a deriver. That is normal.
- Delete via GC, not `rm`.

Try this

1. `nix-store -q --referrers-closure $(nix path-info nixpkgs#hello)` and estimate how much of your profile depends on `hello`.
2. Run `nix-store --verify` without `--check-contents` (fast existence check). Then decide whether you need the slow rehash.
3. `nix path-info --json nixpkgs#hello` and identify `narHash`, `references`, and `deriver` in the JSON.
4. Attempt `nix-store --delete $(nix path-info nixpkgs#glibc)` and read the "cannot delete path because it is still alive" error. That error is the database doing its job.

Garbage Collection

GC Roots and Space Reclamation

GC Roots and Space Reclamation

Nix never overwrites a path, so `/nix/store` grows until something says "this is still needed." The boring default is: **roots keep closures alive; `nix-collect-garbage` deletes everything that is not reachable from a root; `-d` is how you drop old generations on purpose**.

Mental model

A **GC root** is a pointer *outside* the store (or a special symlink inside `gcroots/`) that names a store path. The collector starts from roots and walks `--referrers` in reverse: anything unreachable is garbage.

Common roots:

Root	Typical path	Keeps alive
Current NixOS system	`/run/current-system` and `/nix/var/nix/profiles/system`	The running OS plus old generations
User profile	`~/.nix-profile`	`nix profile` / Home Manager user env
Home Manager generations	`~/.local/state/nix/profiles/home-manager`	Every `home-manager switch` results
`result` symlink	`./result` → registered in `gcroots/auto`	The last `nix-build` in that directory
direnv	`.direnv/flake-profile`	The cached devShell
Channels / flake registry	`~/.nix-defexpr`, registries	Pin data, not usually huge

```
roots  →  profiles / result / current-system

          walk references

          keep those paths; delete the rest
```

`nix-collect-garbage` without `-d` deletes **orphans** (paths with no root) but **keeps generations**. `-d` deletes old generations first, then orphans. That is the difference between "I freed 80 MB" and "I freed 12 GB."

Worked examples

Case 1: See the roots

```
ls /nix/var/nix/gcroots
ls -l /nix/var/nix/gcroots/auto 2>/dev/null | head
nix-store --gc --print-roots 2>/dev/null | head
```

You should recognise `result`, profiles, and `/run/current-system` (on NixOS).

Case 2: A `result` symlink is a root

```
nix build nixpkgs#hello --out-link /tmp/hello-result
nix-store --gc --print-roots | grep hello-result
```

Output:

```
/tmp/hello-result -> /nix/store/…-hello-2.12.1
```

Delete the symlink, then collect:

```
rm /tmp/hello-result
nix-collect-garbage
```

Output:

```
finding garbage collector roots...
deleting garbage...
… store paths deleted, … MiB freed
```

If `hello` is still in your profile, it will **not** be deleted. Only the extra root is gone.

Case 3: Safe collect versus drop generations

```
nix-collect-garbage
```

Keeps every NixOS / Home Manager generation. Safe on a Monday morning.

```
nix-collect-garbage --delete-older-than 14d
```

Drops generations older than fourteen days, then deletes newly orphaned paths. Prefer this over `-d` on a machine you might still want to roll back.

```
sudo nix-collect-garbage -d
```

Drops **all** old generations of all profiles. The running system and the current user profile stay. Rollback targets vanish. Use it when disk is the incident.

On NixOS, save as `gc.nix`:

```
# gc.nix
{
  nix.gc.automatic = true;
  nix.gc.dates = "weekly";
  nix.gc.options = "--delete-older-than 14d";
  nix.settings.auto-optimise-store = true;
  boot.loader.systemd-boot.configurationLimit = 8;
}
```

```
sudo nixos-rebuild switch
systemctl status nix-gc.timer --no-pager
```

That timer is the boring weekly. `-d` remains a human incident command, not the timer.

Case 4: Why CI disks fill up

CI runners that `nix build` without deleting `result`, without a root that expires, and without GC between jobs accumulate every closure they ever substituted. The boring CI pattern:

```
nix build .#desk-api
nix-collect-garbage --delete-older-than 7d
```

or a persistent runner with `auto-optimise-store` and a nightly GC. Ephemeral GitHub runners are destroyed anyway; self-hosted runners are not.

Case 5: Do not GC the system you are running

```
readlink /run/current-system
sudo nix-collect-garbage
ls /run/current-system
```

`/run/current-system` is a root. The collector will not delete the running closure. It **will** delete old generations if you passed `-d`. After `-d`, `nixos-rebuild switch --rollback` has nowhere to go except whatever generations you still have.

The trap

The trap is **`sudo rm -rf /nix/store/* because df is 100%`**. That races the daemon, corrupts `db.sqlite`, and can unboot NixOS. Always GC.

The second trap is running `-d` on a laptop the day before a risky upgrade, then needing the previous generation at 2 a.m. Keep two weeks of generations on machines that boot humans.

The third trap is leaving a `result` in `$HOME` forever. It pins an old clang forever. `find ~ -name result -type l` once a quarter.

The boring rule

- Roots keep closures. No root, GC is allowed to delete.
- `nix-collect-garbage` is the only deletion tool.
- `--delete-older-than 14d` on workstations; `-d` when disk is the incident.
- Delete leftover `result` symlinks you do not need.
- On NixOS, enable automatic GC with a retention window — not `rm`.

Try this

1. `nix build nixpkgs#cowsay --out-link /tmp/cow && nix-store --gc --print-roots | grep /tmp/cow`. Remove the link and GC. Confirm the cowsay path is gone **unless** something else roots it.
2. Count generations: `sudo nix-env --list-generations --profile /nix/var/nix/profiles/system` (NixOS) or `nix-env --list-generations` (user).
3. Dry-run: `nix-collect-garbage --dry-run` and read what would go. Then run without `--dry-run` only if you agree.
4. On a NixOS lab VM, add `nix.gc.automatic = true;` and `nix.gc.dates = "weekly";`, switch, and `systemctl status nix-gc.timer`.

Store Optimization and Deduplication

Store Optimization and Deduplication

Two packages often contain identical files: licenses, headers, man pages, `.pyc` with the same hash. Without help, each copy occupies its own disk blocks. The boring default is: **`nix-store --optimise` (or `nix.optimise.automatic`) hard-links identical files to one inode, and you never do this with `cp` or a dedup FUSE**.

Mental model

Optimisation walks `/nix/store`, hashes file contents, and when two files match:

- They keep two directory entries (two paths).
- They share one inode (one set of blocks).
- They remain read-only. A later rebuild never writes through a hard link; it creates a new path.

```
/nix/store/pkgA/share/COPYING
                                    inode 1001  (one copy on disk)
/nix/store/pkgB/share/COPYING
```

This is **not** content-addressed store paths. Paths stay input-addressed and distinct. Only file *contents* are shared. It is also not compression. `du` after `--optimise` drops; `nix path-info -S` (logical size) may look similar because it counts each path separately.

Nix 2.35 lazy flake copy is eval performance. It is not this. Do not mix them in an incident doc.

Worked examples

Case 1: Run it once and read the report

```
sudo nix-store --optimise
```

Output:

```
hashing files in /nix/store...
31420 files linked, 850.40 MiB freed
```

On a fresh VM the number may be small. On a desk that has compiled three gcc versions it can be gigabytes. Run it a second time immediately: “files linked” near zero. Idempotent.

Case 2: Automatic on NixOS

Save as `optimise.nix` (import from `configuration.nix`):

```
# optimise.nix
{
  nix.optimise.automatic = true;
  nix.optimise.dates = [ "03:45" ];
  nix.settings.auto-optimise-store = true;
}
```

Option	When it runs
`nix.settings.auto-optimise-store`	After **each** build, newly added files are linked
`nix.optimise.automatic`	A systemd timer walks the **whole** store

Turn both on for a workstation. `auto-optimise-store` catches new builds; the timer catches paths that arrived via substitution.

```
sudo nixos-rebuild switch
systemctl status nix-optimise.timer
systemctl list-timers | grep nix
```

On a foreign Linux laptop (Nix 2.35, no NixOS), the same knobs live in the daemon config:

```
# /etc/nix/nix.conf  (or nix.settings via the installer)
auto-optimise-store = true
```

User `~/.config/nix/nix.conf` does not run `--optimise` for daemon builds. Edit the daemon file, then `sudo systemctl restart nix-daemon`.

```
nix show-config | grep auto-optimise
```

`true` means new builds are linked as they land. The weekly timer still walks the whole store for paths that arrived via substitution before the flag was on.

Case 3: Prove two paths share an inode

```
nix build nixpkgs#hello nixpkgs#cowsay
find /nix/store -name COPYING -type f 2>/dev/null | head -5
```

If you have at least two, compare inodes:

```
stat -c '%i %n' /nix/store/<hash-a>-*/COPYING /nix/store/<hash-b>-*/COPYING
```

After `--optimise`, the `%i` values match. `ls -l` shows a link count `n` greater than 1.

```
ls -l /nix/store/<hash-a>-hello-*/share/info/hello.info | awk '{print $2, $NF}'
```

Link count `> 1` means another path shares the inode. That is the disk win. `nix path-info -S` still reports the logical size.

Case 4: Logical size versus disk usage

```
nix path-info -Sh nixpkgs#hello
nix path-info -Shr nixpkgs#hello
df -h /nix
```

`path-info -S` reports the size of that path's files as if they were unique. `df` reports actual blocks after hard links. Both numbers are useful; they are not the same question. When finance asks "how much does hello weigh," use `nix path-info -Shr` (closure, logical). When the disk is full, use `df` and GC.

Case 5: Safe with GC

Optimise, then GC, in either order. Hard links do not keep a path alive. GC deletes directory trees; the inode goes away when the last link is removed.

```
sudo nix-store --optimise
sudo nix-collect-garbage --delete-older-than 14d
df -h /nix
```

That pair is the boring monthly (or the automatic timers). Optimise never deletes a path. If a file vanished, GC or a failed disk did it.

Hard links only work **on one filesystem**. If `/nix/store` is a bind of two disks, `--optimise` cannot share inodes across them. Keep `/nix` on one partition (or one Btrfs/ZFS dataset). Compression (`compress=zstd`) on that dataset is complementary; it is not a substitute for Nix's linker.

The trap

The trap is **a third-party deduplication tool** (`duperemove`, a Btrfs `dedup` cron, zfs `dedup=on` "because Nix") on top of Nix's own hard links. Filesystem-level dedup plus Nix hard links is extra CPU for little gain, and some tools copy-on-write in ways that fight the read-only store.

Btrfs and ZFS compression (`compress=zstd`) is fine. That is not dedup. Leave Nix to hard-link identical files.

```
findmnt /nix || findmnt /
```

Note the filesystem type. `dedup=on` on ZFS plus `nix-store --optimise` is the double-tax. Compression alone is enough beside Nix hard links.

The boring rule

- `nix-store --optimise` or the NixOS timer. Not a random dedup cron.
- Enable `auto-optimise-store` so new builds are linked as they land.
- Measure disk with `df`; measure closures with `nix path-info -Shr`.
- Compression (Btrfs/ZFS) is complementary. Filesystem dedup is not needed.
- Optimise does not replace GC. Run both.

Try this

1. Note `df -h /nix` (or `/`), run `sudo nix-store --optimise`, note `df` again. Record the delta.
2. On NixOS, add the snippet from Case 2, switch, and `systemctl list-timers | grep nix`.
3. `stat` two identical `COPYING` files and confirm the inode number. If you only have one, `nix build nixpkgs#hello nixpkgs#cowsay` first and look again.
4. Run `nix-store --optimise` a second time immediately. The "files linked" count should be near zero. That is how you know it is idempotent.

First Expressions

The Derivation Primitive

The Derivation Primitive

Every package, every system, every `mkShell` eventually becomes a `derivation { ... }`. The boring default is: **know the three required fields, write one by hand once, then use `pkgs.runCommand` / `stdenv.mkDerivation` for the rest of your career**.

Mental model

`derivation` is a builtin. It does not compile C. It produces a `.drv` file describing a build:

Field	Required	Meaning
`name`	yes	Label in the store path
`system`	yes	`"x86_64-linux"`, `"aarch64-linux"`, `"aarch64-darwin"`, …
`builder`	yes	**Store path** of the executable that will run
`args`	no	Arguments to the builder
other attrs	no	Become environment variables inside the sandbox

`$out` is set by Nix to the output path **before** the builder runs. The builder's job is to create that file or directory. If `$out` is missing when the builder exits 0, the build fails.

`builder = "/bin/sh"` is a lie on a sandboxed Linux: `/bin/sh` is not in the sandbox. Point `builder` at `${pkgs.bash}/bin/bash`.

```
derivation { … }

   .drv  (instantiate)

   sandbox runs builder

   $out registered in the store
```

Worked examples

Case 1: A one-file banner

Save as `minimal.nix`:

```
# minimal.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-banner";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [
    "-c"
    "echo 'Desk System Online' > $out"
  ];
}
```

```
nix-build minimal.nix
cat result
```

Output:

```
these 1 derivations will be built:
  /nix/store/…-desk-banner.drv
building '/nix/store/…-desk-banner.drv'...
/nix/store/…-desk-banner
Desk System Online
```

`result` is a GC root pointing at `$out`.

Case 2: Instantiate versus build

```
nix-instantiate minimal.nix
cat $(nix-instantiate minimal.nix) | head
```

The `.drv` is text. You will see `builder`, `system`, and `name`. `nix-build` is instantiate + realise.

Case 3: Environment attributes

Save as `env.nix`:

```
# env.nix
let
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-env";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  deskWindow = "A";
  args = [ "-c" "echo window=$deskWindow > $out" ];
}
```

```
nix-build env.nix && cat result
```

Output:

```
window=A
```

Every extra attribute is an environment variable. Nested attribute sets are **not** allowed here (the builtin is strict about types). That is one reason `stdenv.mkDerivation` exists: it accepts a richer attrset and flattens it.

Case 4: The `runCommand` spelling you will actually write

Save as `run.nix`:

```
# run.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.runCommand "desk-banner" { } ''
  echo 'Desk System Online' > $out
''
```

```
nix-build run.nix && cat result
```

Same `$out` idea. `runCommand` fills in `builder`, `system`, and stdenv so you do not hard-code bash. Prefer this in real files. Use raw `derivation` when you are explaining Nix, not when you are packaging the desk API.

Case 5: Failure when `$out` is missing

Save as `missing.nix`:

```
# missing.nix
let
```

```
  pkgs = import <nixpkgs> {};
in
derivation {
  name = "desk-missing";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [ "-c" "echo forgot" ];
}
```

```
nix-build missing.nix
```

Output:

```
building '/nix/store/…-desk-missing.drv'...
forgot
error: builder for '/nix/store/…-desk-missing.drv' failed to produce output path '/nix/store/…-
```

The builder exited 0. Nix still failed. `$out` is a contract.

`builtins.currentSystem` is **impure**. It is fine in this chapter's throwaway files. A flake must name `system` (`x86_64-linux`, …) so CI and the laptop agree. `passAsFile` exists when an env var would exceed `ARG_MAX`; `stdenv.mkDerivation` uses it so you do not think about it. Multiple outputs (`outputs = [ "out" "dev" ]`) split runtime vs headers — `hello` does not need that; glibc does.

The trap

The trap is **`builder = "/bin/sh"`** copied from an ancient gist. It works only with the sandbox off, or on some macOS setups with a magic `/bin/sh`. On NixOS it fails with `No such file or directory`. Use `${pkgs.bash}/bin/bash` or `runCommand`.

The other trap is writing to `$out` and then also expecting a directory layout without `mkdir`. If `$out` should be a directory, `mkdir -p $out` first; do not echo into `$out` and then `mkdir $out/bin`.

The boring rule

- Raw `derivation` once, so the `.drv` is not magic. Then `runCommand` / `mkDerivation`.
- `builder` is a **store path**, not `/bin/sh`.
- Create `$out`. That is the whole job.
- Extra attributes become env vars. Keep them strings, paths, or lists of those.
- Flakes name `system`; do not ship `builtins.currentSystem`.
- `result` is a GC root. Delete it when you are done poking.

Try this

1. Change the `echo` text in `minimal.nix`, rebuild, and confirm the store path hash changed.
2. `nix-instantiate minimal.nix` and search the `.drv` for `bash`. It must be a `/nix/store/…-bash-…` path.
3. Rewrite `minimal.nix` as `runCommand` (Case 4) and `diff` the two `cat result` outputs — they should match.
4. In `env.nix`, add `deskShift = "morning";` and print both variables. Confirm a rebuild produced a new path.

Building Multi-File Artifacts

Building Multi-File Artifacts

`$out` can be a file or a directory. Real packages are directories: `bin/`, `lib/`, `share/`. The boring default is: **`mkdir -p $out/bin` first, write files, `chmod +x` the executables, and smoke-test with `./result/bin/…`.**

Mental model

Nix does not care about FHS except as a convention other tools expect. If you put a binary at `$out/bin/desk-status`, `nix-env`, Home Manager, and `environment.systemPackages` will put `$out/bin` on `PATH`. If you put it at `$out/desk-status`, they will not.

Path	Usual contents
`$out/bin`	Executables on `PATH`
`$out/lib`	Shared libraries (`RUNPATH` targets)
`$out/share`	Data, man pages, completions
`$out/etc`	Default config (not live `/etc`)

`pkgs.runCommand` is enough for scripts and generated trees. Compilers come later (`stdenv.mkDerivation`).

Worked examples

Case 1: A directory with a script and a data file

Save as `multifile.nix`:

```
# multifile.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.runCommand "desk-assets" { } ''
  mkdir -p $out/bin $out/share
  cat > $out/bin/desk-status << 'EOF'
  #!/bin/sh
  echo status: ready
  EOF
  chmod +x $out/bin/desk-status
  echo 'Desk version 1.0' > $out/share/version.txt
''
```

```
nix-build multifile.nix
./result/bin/desk-status
cat result/share/version.txt
```

Output:

```
/nix/store/…-desk-assets
status: ready
Desk version 1.0
```

Case 2: Patch the shebang so the sandbox shell is not required at runtime

`#!/bin/sh` works on NixOS (there is a `/bin/sh`). It is still sloppy. Point the shebang at store bash:

Save as `shebang.nix`:

```
# shebang.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.runCommand "desk-status" { inherit (pkgs) bash; } ''
  mkdir -p $out/bin
  cat > $out/bin/desk-status << EOF
  #!${pkgs.bash}/bin/bash
  echo status: ready
  EOF
  chmod +x $out/bin/desk-status
''
```

```
nix-build shebang.nix
head -1 result/bin/desk-status
```

Output:

```
#!/nix/store/…-bash-5.2/bin/bash
```

`nix-store -q --references result` now includes bash. That is correct: the script **runs** bash.

Case 3: `writeShellApplication` — the boring wrapper

Save as `write.nix`:

```
# write.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.writeShellApplication {
```

```
  name = "desk-status";
  runtimeInputs = [ pkgs.coreutils ];
  text = ''
    echo "status: ready"
    date -u +%Y-%m-%d
  '';
}
```

```
nix-build write.nix
./result/bin/desk-status
```

Output:

```
status: ready
2026-09-09
```

`writeShellApplication` sets the shebang, `set -o errexit` / `nounset` / `pipefail`, and puts `runtimeInputs` on `PATH` via a wrapper. Prefer it over hand-rolled `echo > $out/bin`.

Case 4: Multiple outputs are a later tool

Some packages split `$out` and `$dev` (headers) so runtime closures stay small. You will meet `outputs = [ "out" "dev" ];` in the packaging part. For desk scripts, one `$out` is enough. Do not invent extra outputs until a closure-size measurement says headers are the problem.

Case 5: Install into a profile to get `PATH`

```
nix-build write.nix
nix profile install ./result
desk-status
```

Output:

```
status: ready
2026-09-09
```

On NixOS, the same package in `environment.systemPackages` is the system-wide equivalent. `./result/bin/…` is the lab check; a profile is how humans run it tomorrow.

The trap

The trap is **forgetting `chmod +x`**. The file is in `$out/bin`, activation puts the directory on `PATH`, and the kernel says `Permission denied`. The build succeeded.

The other trap is writing a shebang of `#!/usr/bin/env bash` and then not putting bash on `PATH` at runtime. `env` looks up `bash` in the **user's** PATH, which may be missing or may be the host's bash. Store-path shebangs (`writeShellApplication`) do not guess.

The boring rule

- `$out` is a directory for anything with more than one file. `mkdir -p` first.
- Executables live in `$out/bin` and are executable.
- Prefer `writeShellApplication` for scripts.
- Shebangs should be store paths, not `/usr/bin/env`.
- Smoke-test `./result/bin/…` before you add the package to a profile.

Try this

1. Add `$out/share/man/man1/desk-status.1` with a one-line man page. Rebuild and `ls result/share/man/man1`.
2. Drop `chmod +x` from Case 1, rebuild, and run `./result/bin/desk-status`. Restore chmod after you see `Permission denied`.
3. `nix-store -q --references $(nix-build shebang.nix --no-out-link)` and find bash. Then do the same for Case 1 (`#!/bin/sh`) and compare.
4. Convert Case 1 to `writeShellApplication` and add `pkgs.hello` to `runtimeInputs`. Call `hello` from the script.

Inspecting the Store

Inspecting Closures: why-depends, path-info, nix-tree

Inspecting Closures: why-depends, path-info, nix-tree

Shipping a "small" service that secretly contains gcc is a weekly desk accident. The boring default is: **measure the closure before you copy it, then ask `why-depends` when a path you did not expect shows up**.

Mental model

Three questions, three tools:

Question	Tool
How big is this, including everything it needs?	`nix path-info -Shr`
Why is **this** path inside **that** closure?	`nix why-depends A B`
Let me walk the tree with a keyboard	`nix-tree`

`nix-store -q --tree` is the classic ASCII dump. `nix-tree` is the same graph with navigation. `nom` (`nix-output-monitor`) wraps `nix build` so you watch fetch/build in a table instead of a scrollback.

You already used `--requisites` in the closures chapter. This chapter is the daily loop: size, blame, browse.

Worked examples

Case 1: Closure size in one line

```
nix path-info -Shr nixpkgs#hello
```

Output (shape):

```
/nix/store/…-glibc-2.39      31.0M  31.0M
/nix/store/…-hello-2.12.1    226K   31.2M
```

The last column on the last line is the number you put in a review: **hello's runtime closure is ~31 MB**, almost all glibc.

Compare a fat wrapper:

```
nix path-info -Sh nixpkgs#ripgrep
nix path-info -Shr nixpkgs#ripgrep | tail -1
```

If the closure is dozens of times the single-path size, something was propagated that should not have been.

Case 2: `why-depends`

"Why does the desk API closure contain python3?"

```
nix why-depends nixpkgs#ripgrep nixpkgs#glibc
```

Output (shape):

```
/nix/store/…-ripgrep-14.1.0
  /nix/store/…-glibc-2.39
```

A longer chain (hello → wrapper → python) prints as a path of edges. That chain is the fix list: cut the edge you did not intend (a `propagatedBuildInputs`, a wrapProgram that pulled in a huge interpreter, a `share/` file that embeds a store path).

If there is **no** dependence:

```
nix why-depends nixpkgs#hello nixpkgs#ripgrep
```

Output:

```
'hello' does not depend on 'ripgrep'
```

Case 3: ASCII tree without extra tools

```
nix-store -q --tree $(nix path-info nixpkgs#hello)
```

Output (shape):

```
/nix/store/…-hello-2.12.1
  /nix/store/…-glibc-2.39
      …
```

Useful on a server where you will not install `nix-tree`. Pipe to `less`.

Case 4: `nix-tree` for interactive blame

```
nix run nixpkgs#nix-tree -- $(nix path-info nixpkgs#hello)
```

or, if `nix-tree` is in your system packages:

```
nix-tree $(nix path-info nixpkgs#hello)
```

Keys (typical):

Key	Action
`j` / `k`	Move
enter	Open a path's references
`s`	Sort by size
`q`	Quit

Sort by size first. The top of the list is where the megabytes are.

Case 5: Watch a build with `nom`

```
nix run nixpkgs#nix-output-monitor -- build nixpkgs#hello
```

`nom` is a frontend. It does not change hashes. Use it when a long `nixos-rebuild` is a wall of identical "copying path" lines and you want to know whether you are substituting or compiling.

On NixOS you can alias:

```
# configuration.nix fragment
environment.systemPackages = [ pkgs.nix-output-monitor ];
```

Then `nom build .#desk-api`.

The trap

The trap is **reading `du -sh result`**. `result` is a symlink to **one** path. `du` does not follow the glibc reference. You report 200 KB, production downloads 31 MB, and someone "optimises" the wrong thing.

Always `nix path-info -Shr`.

The second trap is `why-depends` in the wrong direction. `nix why-depends A B` means "why does A depend on B" — A is the parent closure, B is the unexpected guest.

The boring rule

- Size with `nix path-info -Shr` on the **output** path.
- Blame with `nix why-depends parent child`.
- Browse with `nix-tree` (or `--tree` on a server).
- `du` on `result` is not a closure size.
- `nom` is optional UX. It is not part of the hash.

Try this

1. Record `nix path-info -Shr nixpkgs#hello` and `nix path-info -Shr nixpkgs#git`. Git is larger. `why-depends` git against perl or python and see which interpreter, if any, is in the runtime closure.
2. `nix-store -q --tree $(nix path-info nixpkgs#hello) | wc -l` — that is the node count of a tiny closure.
3. Install `nix-tree` in a devShell, open hello, sort by size, and name the largest path.
4. Run a `nix build` twice, once plain and once under `nom`. Confirm the resulting store path is identical (`nix path-info`).

Nix Language

Nix Expressions and Evaluation

Nix Expressions and Evaluation

Nix does not run a list of statements. It evaluates **one** pure expression. The boring default is: **a file is a single value, the same inputs always yield the same AST, and laziness means unused branches (including `throw`) never run**.

This part uses the Nix **2.35** CLI (`nix eval`). Flakes come in part 3–4; here the language is enough.

Mental model

Bash mutates the world from top to bottom. Nix does not:

1. **One expression per file.** `2 + 2`, `{ a = 1; }`, and a 400-line package are the same kind of thing.
2. **Deterministic.** Same tree, same result. No clock, no `$HOME`, no "it depends who ran it."
3. **Lazy.** Defining fifty attributes and asking for `.port` does not compute the other forty-nine.

```
Bash:   step → mutate → step → mutate → leftover state
Nix:    root expression → demand-driven subexpressions → a value
```

`nix eval --expr` is a scratchpad. `nix eval --file` evaluates a file. `--json` is for machines.

Worked examples

Case 1: Arithmetic is an expression

```
nix eval --expr '1 + 2 * 3'
```

Output:

```
7
```

Precedence is ordinary. There is no `print`. The value *is* the result.

Case 2: A file is one `let … in`

Save as `root_expression.nix`:

```nix
# root_expression.nix
let
  deskId = "terminal-01";
  status = "online";
in
{
  id = deskId;
  state = status;
  ready = true;
}
```

```
nix eval --file root_expression.nix --json
```

Output:

```
{"id":"terminal-01","ready":true,"state":"online"}
```

`--json` sorts keys. The file did not "run" three assignments; it produced one attrset.

Case 3: Laziness skips `throw`

Save as `laziness_proof.nix`:

```nix
# laziness_proof.nix
let
  validField = "desk-service";
  brokenField = throw "This error is never triggered!";
in
{
  service = validField;
  broken = brokenField;
}
```

```
nix eval --file laziness_proof.nix --apply 'x: x.service'
```

Output:

```
"desk-service"
```

`broken` exists on the set. It is not forced. Force it:

```
nix eval --file laziness_proof.nix --apply 'x: x.broken'
```

Output (shape):

```
error: This error is never triggered!
```

That error is the proof: evaluation is demand-driven.

Case 4: `--strict` / JSON vs thunks

```
nix eval --expr '{ a = 1 + 1; b = throw "no"; }' --apply 'x: x.a'
```

Output:

```
2
```

`nix-instantiate --eval` without `--strict` can print thunks as `<CODE>`. Prefer `nix eval` (2.35) plus `--json` or `--apply` so you see values, not internals.

Case 5: No side effects at eval time

Save as `no_echo.nix`:

```
# no_echo.nix
let
  # This does not print. It is a string.
  msg = "desk online";
in
msg
```

```
nix eval --file no_echo.nix
```

Output:

```
"desk online"
```

There is no `echo` during eval. Writes happen later, in a **derivation** sandbox (fundamentals / packaging). If you wanted a file on disk, you have not evaluated yet — you have not built.

`throw "msg"` is a lazy error (only if demanded). `abort "msg"` is immediate. Use `throw` in unused branches; `abort` when the file must not even load. Nix 2.35 can drop you into a debugger:

```
nix eval --file laziness_proof.nix --apply 'x: x.broken' --debugger
```

`:backtrace` then. Do not leave `builtins.trace` in production modules after you found the bug.

The trap

The trap is **expecting `echo` or `curl` inside a `.nix` file to run when you `nix eval`**. Eval is memory. Network and files belong in FOD fetchers and builders.

A second trap: two expressions in one file with no `let`. That is a parse error. One root.

The boring rule

- One expression per file.
- `nix eval --expr` / `--file` on Nix 2.35. `--json` when you want stable output.
- Laziness: unused `throw` is silent; accessed `throw` is an eval error. `abort` always fires.
- `--debugger` on 2.35 for eval traces. No leftover `builtins.trace` on main.
- No side effects at eval. Builds write `$out`.
- `--apply` to pick a field without evaluating the whole set.

Try this

1. Case 3: eval `.service` then `.broken`. Predict before you run.
2. `nix eval --expr 'let x = 1 / 0; in 4'` — does it error? (Integer division by zero is forced only if `x` is used.)
3. Put two attrsets in one file without `let`. Read the parse error.
4. `nix eval --expr '{ a = 1; }' --json` and compare to `--impure` — you should not need `--impure` here.

Values and Immutability

Values and Immutability

Once a Nix value exists, nothing mutates it. The boring default is: **`//` and `++` allocate new values; names in a `let` are bound once; the original attrset is still what you passed in**.

Mental model

Python dicts surprise you:

```
config = {"port": 8080}
mutate(config)  # original changed
```

Nix:

1. Attrsets, lists, and strings do not have setters.
2. `a // b` is a **new** set. `a` is unchanged.
3. Memoisation is safe because nothing behind a pointer can change.

```
{ a = 1; }
                { a = 1; b = 2; }   (new)
{ b = 2; }
first set still { a = 1; }
```

Worked examples

Case 1: Merge does not rewrite

Save as `immutable_merge.nix`:

```
# immutable_merge.nix
let
  baseDesk = {
    location = "building-a";
    tier = "standard";
  };
  upgradedDesk = baseDesk // {
    tier = "executive";
    desksCount = 12;
```

```
  };
in
{
  original = baseDesk;
  upgraded = upgradedDesk;
}
```

```
nix eval --file immutable_merge.nix --json
```

Output:

```
{"original":{"location":"building-a","tier":"standard"},"upgraded":{"desksCount":12,"location":
```

`baseDesk.tier` is still `"standard"`. Fifty functions can receive `baseDesk`.

Case 2: Lists concatenate to a new list

Save as `immutable_list.nix`:

```
# immutable_list.nix
let
  activeTables = [ 1 2 3 ];
  extendedTables = activeTables ++ [ 4 5 ];
in
{
  initial = activeTables;
  result = extendedTables;
}
```

```
nix eval --file immutable_list.nix --json
```

Output:

```
{"initial":[1,2,3],"result":[1,2,3,4,5]}
```

There is no `.push`.

Case 3: Same name twice is an error

Save as `double_bind.nix`:

```
# double_bind.nix
let
  x = 1;
```

```
  x = 2;
in
x
```

```
nix eval --file double_bind.nix
```

Output (shape):

```
error: attribute 'x' already defined at double_bind.nix:3
```

Use a second name (`x0`, `xNext`) or an inner `let` that **shadows**.

Case 4: Inner `let` shadows, outer unchanged

Save as `shadow.nix`:

```
# shadow.nix
let
  x = 1;
in
{
  outer = x;
  inner = let x = 2; in x;
}
```

```
nix eval --file shadow.nix --json
```

Output:

```
{"inner":2,"outer":1}
```

Case 5: Deep merge is not `//`

```
# shallow.nix
let
  a = { desk = { port = 80; host = "a"; }; };
  b = { desk = { port = 443; }; };
in
a // b
```

```
nix eval --file shallow.nix --json
```

Output:

```
{"desk":{"port":443}}
```

`host` is **gone**. `//` replaces the `desk` attr wholesale. Nested merge is `lib.recursiveUpdate` (imports chapter) or a function that rebuilds the nest. Do not expect `//` to patch deep keys.

The trap

The trap is `let x = 1; x = 2;`. The trap after that is assuming `//` is deep. Both look like assignment in other languages. They are not.

The boring rule

- Values do not change. New names, new values.
- `//` shallow. `++` new list.
- One binding per name per `let`.
- Shadow in a nested `let`, do not reassign.
- For nested config, write a function or `lib.recursiveUpdate`, not a pile of `//`.

Try this

1. `nix eval --expr 'let a = { x = 1; }; b = a // { x = 2; }; in a.x'` — expect `1`.
2. Case 5: add `lib` later; for now rebuild `{ desk = a.desk // b.desk; }` by hand and keep `host`.
3. Duplicate a `let` name; save the error text.
4. `nix eval --expr '[ 1 ] ++ [ 1 ]'` — duplicates are allowed; lists are not sets.

Data Types and String Interpolation

Data Types and String Interpolation

Nix checks types when a value is **used**, not at parse time. The boring default is: **know the six primitives, interpolate with `${…}`, `toString` numbers, and treat unquoted paths as store copies**.

Mental model

Type	Examples
`int`	`42`, `-1`
`float`	`3.14`
`bool`	`true`, `false`
`string`	`"desk"`, `''multi''`
`path`	`./config.nix`, `/etc/os-release`
`null`	`null`

Also later: lists, attrsets, functions, derivations.

`"${port}"` is an error if `port` is an int. `"${toString port}"` is the boring spelling.

`'' … ''` is a multi-line string. Leading indentation of the closing `''` is stripped.

Unquoted `./file` interpolated into a string **copies** `file` into `/nix/store` and pastes that path. Quoted `"./file"` is just characters.

Worked examples

Case 1: `builtins.typeOf`

Save as `types_check.nix`:

```nix
# types_check.nix
{
  intType = builtins.typeOf 100;
  floatType = builtins.typeOf 3.5;
  boolType = builtins.typeOf true;
  stringType = builtins.typeOf "desk";
  pathType = builtins.typeOf ./types_check.nix;
  nullType = builtins.typeOf null;
```

```
}
```

```
nix eval --file types_check.nix --json
```

Output:

```
{"boolType":"bool","floatType":"float","intType":"int","nullType":"null","pathType":"path","str
```

Case 2: Interpolation and indented strings

Save as `interpolation.nix`:

```
# interpolation.nix
let
  appName = "desk-agent";
  port = 9090;
  logDir = "/var/log/desk";
  endpoint = "http://127.0.0.1:${toString port}";
  systemdService = ''
    [Unit]
    Description=${appName} Service

    [Service]
    ExecStart=/bin/false
    Environment=ENDPOINT=${endpoint}
    StandardOutput=append:${logDir}/output.log
  '';
in
{
  inherit endpoint systemdService;
}
```

```
nix eval --file interpolation.nix --apply 'x: x.endpoint'
```

Output:

```
"http://127.0.0.1:9090"
```

```
nix eval --file interpolation.nix --apply 'x: x.systemdService'
```

The unit text has no leftover leading spaces from the Nix indent. That is why `''` exists for systemd snippets.

Case 3: Path vs string

```
nix eval --expr 'let f = ./types_check.nix; in "${f}"'
```

Output (shape):

```
"/nix/store/…-types_check.nix"
```

```
nix eval --expr '"./types_check.nix"'
```

Output:

```
"./types_check.nix"
```

The first copied bytes into the store (eval of a path). The second is a string you could pass to a builder as a **name**, not as content.

Case 4: Integer interpolation fails

Save as `bad_interp.nix`:

```
# bad_interp.nix
let port = 8080; in "listen ${port}"
```

```
nix eval --file bad_interp.nix
```

Output (shape):

```
error: cannot coerce an integer to a string
```

Fix: `"listen ${toString port}"`.

Case 5: `null` is not `""`

```
nix eval --expr 'builtins.typeOf null'
nix eval --expr 'null == ""'
```

Output:

```
"null"
false
```

`or` on attrsets (`set.x or 3`) is not the same as `null`. Missing attr null attr.

The trap

The trap is **quoting a path** (`src = "./src"`) and wondering why the builder cannot see your files. Unquoted `./src` is the FOD/source. A string is letters.

The other trap is interpolating a derivation into a string at **eval** (`"${pkgs.hello}"`) — that **forces a build** (or at least a path). Fine in a module; surprising in a "just eval" scratch file.

The boring rule

- `toString` before interpolating ints (and most non-strings).
- `''…''` for scripts and units.
- Unquoted paths when Nix should hash the file.
- Quoted paths when you mean a literal string.
- `builtins.typeOf` when a merge "looks like" the wrong kind.

Try this

1. `nix eval --expr 'builtins.typeOf (toString 42)'` → `"string"`.
2. Case 4: add `toString`, re-eval.
3. `nix eval --expr 'let p = 1; in "${p}"'` and save the coerce error.
4. Compare `"${./types_check.nix}"` with `toString ./types_check.nix` — both store paths, slightly different spelling rules; pick one style and keep it.

Let-In Blocks and Local Bindings

Let-In Blocks and Local Bindings

There is no `x = 1` that you later change. The boring default is: **`let` names, `in` uses them, bindings may mention each other in any order, and unused names (even `throw`) stay cheap**.

Mental model

```
let
  b = a + 10;
  a = 5;
in
b   # 15
```

Order in the `let` is not execution order. Nix builds a graph. `b` needs `a`; `a` is `5`.

`inherit x;` in an attrset is `x = x;` — copy the name from scope.

Nested `let` shadows. The outer name is still there for the outer `in`.

Worked examples

Case 1: Desk URL from parts

Save as `desk_service.nix`:

```
# desk_service.nix
let
  domain = "internal.desk";
  port = 443;
  protocol = "https";
  serviceUrl = "${protocol}://${domain}:${toString port}";
in
{
  inherit domain port;
  url = serviceUrl;
}
```

```
nix eval --file desk_service.nix --json
```

Output:

```
{"domain":"internal.desk","port":443,"url":"https://internal.desk:443"}
```

Case 2: Order does not matter

Save as `order.nix`:

```nix
# order.nix
let
  c = b + 1;
  b = a + 1;
  a = 1;
in
c
```

```
nix eval --file order.nix
```

Output:

```
3
```

Write names in dependency order anyway. Humans read top-down even when Nix does not.

Case 3: Shadowing

Save as `scope.nix`:

```nix
# scope.nix
let
  tier = "standard";
in
{
  outerTier = tier;
  overridden =
    let
      tier = "premium";
    in
    tier;
}
```

```
nix eval --file scope.nix --json
```

Output:

```
{"outerTier":"standard","overridden":"premium"}
```

Case 4: Unused `throw` is fine

Save as `unused.nix`:

```
# unused.nix
let
  domain = "internal.desk";
  unused = throw "never evaluated";
in
domain
```

```
nix eval --file unused.nix
```

Output:

```
"internal.desk"
```

A 200-line `let` of unused helpers is still a readability problem. Laziness does not make a mess cheap for the next reader.

Case 5: `inherit` from a set

Save as `inherit_from.nix`:

```
# inherit_from.nix
let
  cfg = { host = "desk"; port = 8080; extra = true; };
in
{
  inherit (cfg) host port;
}
```

```
nix eval --file inherit_from.nix --json
```

Output:

```
{"host":"desk","port":8080}
```

`extra` stayed in `cfg`. `inherit (cfg) host port` is the boring way to pick keys.

The trap

The trap is **a 200-line `let` of unrelated helpers**. Split files (`import`, next chapters) when a binding cluster has a name (`db`, `tls`, `desk-api`).

The other trap is circular `let`: `a = b; b = a;`. Eval loops. `rec` attrsets have the same foot-gun (next chapter).

The boring rule

- `let` for names. `in` for the value you mean.
- Bindings can be in any order; write them in human order.
- `inherit` / `inherit (set)` instead of `host = host`.
- Unused bindings can `throw`; still delete them if they confuse.
- Split large `let`s into files.

Try this

1. Add `unused = throw "x";` to `desk_service.nix`; eval still works.
2. Reverse three bindings as in Case 2; eval.
3. `inherit (cfg) missing;` and read the error.
4. Nested `let` that shadows `port`; print both outer and inner.

Functions and Argument Defaults

Functions and Argument Defaults

A Nix function takes **one** argument. The boring default for desk config is: **`{ name, port ? 8080, ... }:` — named attrs, defaults with `?`, `...` only when extras are allowed**.

Mental model

```
x: x + 1
```

Two "parameters" are currying: `x: y: x + y`, called as `add 1 2`. Configuration code should not look like that.

Named args:

```
{ host, port ? 80 }: "${host}:${toString port}"
```

Piece	Meaning
`{ host, port }`	Both required
`port ? 80`	Optional
`...`	Ignore extra attrs
`args@{ host, ... }`	Bind the whole set as `args`

Without `...`, an unexpected key is an eval error. That is often what you want.

Worked examples

Case 1: Curry vs attrset

Save as `functions_demo.nix`:

```
# functions_demo.nix
let
  add = x: y: x + y;
  formatServer = { host, port ? 80 }: "${host}:${toString port}";
in
{
  sum = add 10 25;
  defaultPort = formatServer { host = "api.desk"; };
```

```
  customPort = formatServer { host = "api.desk"; port = 443; };
}
```

```
nix eval --file functions_demo.nix --json
```

Output:

```
{"customPort":"api.desk:443","defaultPort":"api.desk:80","sum":35}
```

Call `formatServer` with a missing `host` — error. That is better than `"null:80"`.

Case 2: `@` captures extras

Save as `capture.nix`:

```
# capture.nix
let
  makeProfile = args@{ user, role ? "member", ... }: {
    username = user;
    userRole = role;
    rawInput = args;
  };
in
makeProfile { user = "alice"; team = "desk-ops"; }
```

```
nix eval --file capture.nix --json
```

Output:

```
{"rawInput":{"team":"desk-ops","user":"alice"},"userRole":"member","username":"alice"}
```

`team` survived in `rawInput` because of `...`.

Case 3: Unexpected argument without `...`

Save as `strict_fn.nix`:

```
# strict_fn.nix
let
  f = { host }: host;
in
f { host = "desk"; extra = 1; }
```

```
nix eval --file strict_fn.nix
```

Output (shape):

```
error: function 'anonymous lambda' called with unexpected argument 'extra'
```

Package functions (`{ stdenv, fetchurl }:`) stay strict so typos fail. Module-shaped `{ config, pkgs, ... }:` uses `...` because the module system passes extra keys.

Case 4: `secure ? false`

Save as `url.nix`:

```
# url.nix
let
  formatServer = { host, port ? 80, secure ? false }:
    let
      scheme = if secure then "https" else "http";
      p = if secure && port == 80 then 443 else port;
    in
    "${scheme}://${host}:${toString p}";
in
{
  a = formatServer { host = "api.desk"; };
  b = formatServer { host = "api.desk"; secure = true; };
}
```

```
nix eval --file url.nix --json
```

Output:

```
{"a":"http://api.desk:80","b":"https://api.desk:443"}
```

Case 5: Apply with `--apply`

```
nix eval --file functions_demo.nix --apply 'x: x.customPort'
```

Output:

```
"api.desk:443"
```

Functions are values. You can pass them around; you cannot mutate a default after the fact — you call again with a different set.

The trap

The trap is `...` **on every function** so "it never errors." Typos (`prt = 443`) silently drop. Use `...` at module boundaries. Keep package headers strict.

The other trap is currying five levels (`a: b: c: d: e:`) for config. Named attrsets.

`{ pkgs, ... }@args:` keeps extras **and** names. `{ pkgs }:` rejects unknown keys. `{ pkgs, lib ? pkgs.lib, ... }:` is the usual package/module header. `@args` is for `args.config` forwarding, not for ignoring typos — still list the keys you read.

The boring rule

- Config functions take `{ … }`.
- `?` for defaults. Required keys stay required.
- `...` only when extras are part of the contract.
- `@args` when you must forward the whole set.
- Unexpected-argument errors are a feature.

Try this

1. Case 4: add `path ? ""` and append it when non-empty.
2. Drop `...` from Case 2 and pass `team`; read the error.
3. `nix eval --expr '({ x }: x) { x = 1; y = 2; }'`.
4. Write `id = x: x;` and `nix eval --expr '(import ./id.nix) 41'` after saving `id.nix` as `x: x`.

Attribute Sets and Recursive Definitions

Attribute Sets and Recursive Definitions

Attrsets are how Nix holds config. The boring default is: **plain `{ }`, merge with `//`, read with `.` and `or`, and use `rec` only when siblings need each other and you will not `//` over them later**.

Mental model

```
{ host = "127.0.0.1"; port = 8080; }
```

Op	Role
`s.port`	Access; missing → error
`s.timeout or 30`	Default if missing
`a // b`	Shallow merge; right wins
`rec { x = 1; y = x; }`	Siblings see each other

`desk.db.port = 5432;` in a `let` bind is nested sugar in some positions (`nested` in `rec` / `let` via `x.y =`). Prefer nested `{ desk.db = { port = 5432; }; }` when learning.

Worked examples

Case 1: Merge and `or`

Save as `attr_operations.nix`:

```
# attr_operations.nix
let
  defaultConfig = {
    host = "127.0.0.1";
    port = 8080;
    workers = 4;
  };
  productionOverrides = {
    port = 443;
    workers = 16;
  };
  finalConfig = defaultConfig // productionOverrides;
```

```
in
{
  inherit finalConfig;
  customTimeout = finalConfig.timeout or 30;
}
```

```
nix eval --file attr_operations.nix --json
```

Output:

```
{"customTimeout":30,"finalConfig":{"host":"127.0.0.1","port":443,"workers":16}}
```

Case 2: `rec` for sibling URLs

Save as `rec_demo.nix`:

```
# rec_demo.nix
rec {
  domain = "desk.corp";
  authService = "https://auth.${domain}";
  billingService = "https://billing.${domain}";
  endpoints = [ authService billingService ];
}
```

```
nix eval --file rec_demo.nix --apply 'x: x.authService'
```

Output:

```
"https://auth.desk.corp"
```

A `let` around the same names is often clearer than `rec`. Use `rec` when the set *is* the value you export.

Case 3: `//` does not update `rec` dependents

Save as `rec_override.nix`:

```
# rec_override.nix
let
  base = rec { x = 1; y = x + 1; };
  updated = base // { x = 10; };
in
{
  inherit (updated) x y;
}
```

```
nix eval --file rec_override.nix --json
```

Output:

```
{"x":10,"y":2}
```

`y` is still 2. It closed over the original `x`. If overrides must recompute, use a function:

```
make = { x }: rec { inherit x; y = x + 1; };
make { x = 10; }  # y = 11
```

Case 4: Missing attr without `or`

```
nix eval --file attr_operations.nix --apply 'x: x.finalConfig.timeout'
```

Output (shape):

```
error: attribute 'timeout' missing
```

Production config: `or` at the edges, required keys in the module types (part 6).

Case 5: `//` is shallow (again)

```
# nest.nix
let
  a = { tls = { enable = true; port = 443; }; };
  b = { tls = { enable = false; }; };
in
a // b
```

```
nix eval --file nest.nix --json
```

Output:

```
{"tls":{"enable":false}}
```

`port` vanished. Rebuild `tls = a.tls // b.tls` or `lib.recursiveUpdate`.

```
# nest-fix.nix
let
  lib = import <nixpkgs/lib>;
  a = { tls = { enable = true; port = 443; }; };
```

```
  b = { tls = { enable = false; }; };
in
lib.recursiveUpdate a b
```

```
nix eval --impure --file nest-fix.nix --json
```

```
{"tls":{"enable":false,"port":443}}
```

`recursiveUpdate` is still not the module system (no `mkForce`). Use it for data. Use modules for NixOS config.

The trap

The trap is **`rec` plus `//` and expecting dependents to move**. Case 3. Prefer `let` + function.

The other trap is `s.foo.bar` when `foo` might be missing — error on `foo`, not on `bar`. `s.foo.bar or 1` does not save you; `or` binds the whole `s.foo.bar` only if the syntax is `(s.foo.bar) or 1` vs `s.foo.bar or 1` — in Nix, `or` is tight on the selection. Write `(s.foo or {}).bar or 1` when nested.

The boring rule

- `.` and `or` for reads. `//` for shallow overlays of config.
- `rec` for small self-contained sets you will not patch.
- Overrides that must recompute → function, not `rec //`.
- Nested data: merge one level at a time or `recursiveUpdate`.
- Missing attributes should error in internal code; `or` at user-facing edges.

Try this

1. Access `finalConfig.invalid` without `or`.
2. Rewrite `rec_demo.nix` as `let domain = …; in { … }` without `rec`.
3. Case 3: replace `//` with `make { x = 10; }` and confirm `y == 11`.
4. `(s.tls or {}).port or 443` on a set without `tls`.

Conditionals, Lists, and Built-in Operations

Conditionals, Lists, and Built-in Operations

`if` in Nix is a **value**, not a statement. The boring default is: **`if c then a else b` (`else` required), lists with `++`, and `builtins.map` / `filter` / `length` / `elem`.**

Mental model

```
if isProd then 5 else 1
```

No `if` without `else`. No `if` that "does" something. Both branches are expressions; the unused one stays lazy.

Lists:

```
[ 1 "two" ./three (4 + 5) ]
```

Elements are space-separated. `[ 1 2 ] ++ [ 3 ]` is `[ 1 2 3 ]`. Indexing is `builtins.elemAt xs 0` (0-based). There is no `xs[0]` in the language.

`lib.optional cond x` is `if cond then [ x ] else [ ]` — the boring way to maybe-append one item.

Worked examples

Case 1: Flags

Save as `conditional_config.nix`:

```
# conditional_config.nix
let
  isProduction = true;
in
{
  replicas = if isProduction then 5 else 1;
  logging = if isProduction then "warn" else "debug";
}
```

```
nix eval --file conditional_config.nix --json
```

Output:

```
{"logging":"warn","replicas":5}
```

Flip `isProduction` to `false` and eval again.

Case 2: map / filter

Save as `list_operations.nix`:

```nix
# list_operations.nix
let
  tableNumbers = [ 1 2 3 4 5 6 7 8 9 10 ];
  isPriority = n: n <= 3;
  priorityTables = builtins.filter isPriority tableNumbers;
  deskWorkers = builtins.map (n: { table = n; status = "active"; }) priorityTables;
in
{
  inherit deskWorkers;
  n = builtins.length priorityTables;
  hasTen = builtins.elem 10 tableNumbers;
}
```

```
nix eval --file list_operations.nix --json
```

Output:

```
{"deskWorkers":[{"status":"active","table":1},{"status":"active","table":2},{"status":"active",
```

Case 3: `else` is required

Save as `bad_if.nix`:

```nix
# bad_if.nix
if true then 1
```

```
nix eval --file bad_if.nix
```

Output (shape):

```
error: syntax error, unexpected end of file, expecting 'else'
```

To omit an attr, do not skip `else`. Use:

```nix
{ } // (if enable then { workers = 4; } else { })
```

or `lib.optionalAttrs enable { workers = 4; }`.

Case 4: Concatenate packages-style lists

Save as `packages.nix`:

```
# packages.nix
let
  common = [ "git" "jq" ];
  extra = [ "ripgrep" ];
in
common ++ extra ++ (if true then [ "htop" ] else [ ])
```

```
nix eval --file packages.nix --json
```

Output:

```
["git","jq","ripgrep","htop"]
```

This is how `environment.systemPackages` concatenates across modules (the module system merges lists). Here you do it by hand.

Case 5: Lazy unused branch

Save as `lazy_if.nix`:

```
# lazy_if.nix
if true then "ok" else throw "no"
```

```
nix eval --file lazy_if.nix
```

Output:

```
"ok"
```

The `throw` is not forced. Invert the condition to see it.

The trap

The trap is **`if cond then x;`** with no **`else`**, copied from Bash. Parse error.

The other trap is `[ 1, 2 ]` with commas. Nix lists are **spaces**. Commas are attrset / function-arg style. `[ 1, 2 ]` is often a parse error or a surprising function call.

The boring rule

- `if` always has `else`. Both sides are values.
- Spaces in lists. `++` to join.
- `map` / `filter` / `length` / `elem` from `builtins`.
- `optional` / `optionalAttrs` (lib) for maybe-one-item.
- Unused `else throw` stays quiet — good for flags, bad if you meant to test both.

Try this

1. `builtins.length` already in Case 2; add `builtins.elem 3 priorityTables`.
2. Write `[ 1, 2 ]` and read the error (or the odd parse).
3. Case 5 with `false`.
4. `lib.optional` — after you have `pkgs.lib` in the imports chapter, replace the `if true then [ "htop" ] else [ ]` pattern.

Importing Files and Building Abstractions

Importing Files and Building Abstractions

One file does not hold a desk. The boring default is: **small files that return functions, `import ./x.nix { … }` with explicit args, and no `<nixpkgs>` in anything you ship**.

Mental model

`import` evaluates a path to a value:

```
import ./constants.nix
import ./service.nix { port = 8080; }
```

The imported file cannot see your `let` names unless you pass them. There is no implicit global `pkgs` unless you created one.

`<nixpkgs>` looks up `NIX_PATH`. Two laptops, two channels, two worlds. Flakes pin `github:NixOS/nixpkgs/nixos-2` This chapter still uses relative `./` files so you see the operator.

`pkgs.lib` (once you have pkgs) is `optional`, `mkIf`, `recursiveUpdate`, `toUpper`, … — prefer it over reinventing `if cond then [ x ] else [ ]`.

Paths in `import ./foo.nix` are relative to **the file that contains the import**, not to the shell cwd.

Worked examples

Case 1: Function file + caller

Save as `database.nix`:

```
# database.nix
{ environment }:

let
  isProduction = environment == "production";
in
{
  dbHost = if isProduction then "db.desk.corp" else "127.0.0.1";
  dbPort = if environment == "staging" then 5433 else 5432;
  poolSize = if isProduction then 50 else 5;
```

```
}
```

Save as `app.nix`:

```
# app.nix
let
  dbConfig = import ./database.nix { environment = "production"; };
in
{
  appName = "desk-billing";
  database = dbConfig;
}
```

```
nix eval --file app.nix --json
```

Output:

```
{"appName":"desk-billing","database":{"dbHost":"db.desk.corp","dbPort":5432,"poolSize":50}}
```

Case 2: Staging port

```
nix eval --expr '(import ./database.nix { environment = "staging"; }).dbPort'
```

Output:

```
5433
```

Missing arg:

```
nix eval --expr 'import ./database.nix {}'
```

```
error: function 'anonymous lambda' called without required argument 'environment'
```

That error is the boring default working. Make the arg optional only when you mean it: `{ environment ? "dev" }:`.

Case 3: `lib.optional`

Save as `lib_demo.nix`:

```
# lib_demo.nix
let
  optional = cond: x: if cond then [ x ] else [ ];
```

```
  features = [ "core-api" ]
    ++ optional true "metrics-exporter"
    ++ optional false "debug-profiler";
in
features
```

```
nix eval --file lib_demo.nix --json
```

Output:

```
["core-api","metrics-exporter"]
```

With nixpkgs 26.05: `lib.optional` is the same idea. Do not copy half of `lib` into every repo; `pkgs.lib` once you import nixpkgs.

Case 4: Relative paths are relative to the file

`import ./database.nix` in `app.nix` looks next to `app.nix`, not next to your shell cwd. `nix eval --file ./app.nix` from `/tmp` still finds `./database.nix` beside `app.nix`.

```
cd /tmp
nix eval --file /path/to/desk/app.nix --json
```

Same JSON as Case 1.

Case 5: `<nixpkgs>` is ambient

```
nix eval --impure --expr 'builtins.nixPath'
nix eval --impure --expr '<nixpkgs>'
```

Whatever `nixpkgs=` points at is **not** in `flake.lock`. Fine for a 10-second repl. Not fine for the desk flake.

```
# do not ship
{ pkgs ? import <nixpkgs> { } }: pkgs.hello
```

Next parts: `inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";`. Circular `import` (`a.nix` imports `b.nix` imports `a.nix`) is an eval loop — pass data downward.

The trap

The trap is **`import <nixpkgs> {}` in production modules**. CI, a coworker, and a 6-month-old laptop disagree. Pin 26.05 in a lockfile.

The other trap is circular `import`. Eval loop. Pass data downward; do not make a cycle.

The boring rule

- Files return functions. Callers pass args.
- `import ./relative.nix` — path relative to the file, not cwd.
- No `<nixpkgs>` in shipped config. Flake pin `nixos-26.05`.
- Use `pkgs.lib` instead of a private util soup — once pkgs exists.
- No import cycles.

Try this

1. `import ./database.nix { environment = "staging"; }` and check `dbPort == 5433`.
2. Move `app.nix` eval to another cwd; confirm it still works.
3. `nix eval --impure --expr '<nixpkgs>'` and note it is a path, not a lock.
4. Split a `let` from an earlier chapter into `url.nix` + `import`.

The Nix REPL and Debugging Expressions

The Nix REPL and Debugging Expressions

The boring default before editing any Nix file is **opening `nix repl .#` first**. It gives you an interactive evaluator for the current flake, lets you inspect types and attribute paths without touching the file system, and surfaces errors instantly — the same role Python's REPL or `ghci` plays for Haskell.

Mental model

`nix repl` is an incremental expression evaluator. Each line you type is parsed and evaluated; the result is printed. The session accumulates bindings: assign with `x = 42` and `x` stays in scope for the rest of the session.

Key REPL meta-commands:

Command	What it does
`:t expr`	Print the inferred type of `expr`
`:e expr`	Open the definition of `expr` in `$EDITOR`
`:b drv`	Realise a derivation (`nix-store --realise`)
`:p expr`	Pretty-print a value, expanding lazy thunks
`:l path`	Load a `.nix` file or flake into scope
`:r`	Reload — re-evaluate all `:l` expressions
`:?`	List every meta-command

There are two common launch modes:

```
nix repl                    # empty scope
nix repl 'nixpkgs'          # loads nixpkgs; pkgs = import <nixpkgs> {}
nix repl .#                 # loads the flake in the current directory
```

Nix evaluation is **lazy**. The REPL only evaluates what you ask for. An attribute set with a broken nested path will not error until you actually access that path.

Worked examples

Case 1: Basic REPL session

This walkthrough covers arithmetic, strings, attribute sets, and function application — the primitives you will inspect constantly at the infrastructure desk.

Start the REPL with no arguments:

```
nix repl
```

Output:

```
Nix 2.35.2
Type :? for help.

nix-repl>
```

Arithmetic:

```
nix-repl> 2 + 2
4

nix-repl> 1024 * 1024
1048576

nix-repl> 7 / 2
3
```

Division is integer division for two integers. Use floats explicitly if you need fractional results.

Strings and interpolation:

```
nix-repl> "desk-" + "api"
"desk-api"

nix-repl> host = "db.desk.corp"

nix-repl> "connecting to ${host}:5432"
"connecting to db.desk.corp:5432"
```

Attribute sets:

```
nix-repl> svc = { name = "billing"; port = 8080; }

nix-repl> svc.name
"billing"

nix-repl> svc.port
8080
```

Function application:

```
nix-repl> greet = name: "Hello, ${name}!"

nix-repl> greet "ops-team"
"Hello, ops-team!"
```

Type introspection with `:t`:

```
nix-repl> :t 42
an integer

nix-repl> :t "desk-api"
a string

nix-repl> :t svc
a set

nix-repl> :t greet
a function

nix-repl> :t [ 1 2 3 ]
a list

nix-repl> :t true
a Boolean
```

`:t` is the fastest way to catch a "I expected a string but got a set" mismatch before it becomes a build error.

Case 2: Loading nixpkgs

`nix repl 'nixpkgs'` imports nixpkgs and places its attribute set into scope. This is the standard way to inspect package metadata, library functions, and attribute paths at the infrastructure desk.

```
nix repl 'nixpkgs'
```

Output:

```
Nix 2.35.2
Type :? for help.

Loading installable 'nixpkgs'...
Added 22756 variables.

nix-repl>
```

Query a package version:

```
nix-repl> go.version
"1.22.3"

nix-repl> postgresql.version
"16.3"

nix-repl> prometheus.version
"2.51.2"
```

Inspect package metadata:

```
nix-repl> go.meta.description
"The Go Programming Language"

nix-repl> go.meta.homepage
"https://go.dev/"
```

Browse `lib` functions. Every function under `lib` is an attribute — tab-complete works:

```
nix-repl> lib.strings.toUpper "hello"
"HELLO"

nix-repl> lib.strings.concatStringsSep ", " [ "api" "worker" "scheduler" ]
"api, worker, scheduler"

nix-repl> lib.lists.flatten [ [ 1 2 ] [ 3 [ 4 5 ] ] ]
[ 1 2 3 4 5 ]

nix-repl> lib.attrsets.mapAttrs (k: v: v + 1) { a = 1; b = 2; }
{ a = 2; b = 3; }
```

Jump to a function's source with `:e`:

```
nix-repl> :e lib.strings.concatStringsSep
```

This opens `lib/strings.nix` at the definition of `concatStringsSep` in your `$EDITOR`. It is the fastest way to understand what a library function actually does without searching the nixpkgs repository manually.

Check whether an attribute exists before using it:

```
nix-repl> lib.attrsets.hasAttr "version" go
true

nix-repl> lib.attrsets.hasAttr "doesNotExist" go
false
```

Case 3: Loading a flake

`nix repl .#` loads the flake in the current directory. Every top-level output — `packages`, `devShells`, `nixosConfigurations`, `apps` — becomes directly accessible in scope.

The infrastructure desk flake used throughout this section:

```
# flake.nix
{
  description = "Desk infrastructure flake";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      packages.${system}.default = pkgs.writeShellApplication {
        name = "desk-hello";
        text = ''
          echo "Infrastructure desk tooling - online."
        '';
      };

      devShells.${system}.default = pkgs.mkShell {
        packages = [ pkgs.go pkgs.gopls pkgs.golangci-lint ];
        shellHook = ''
```

```
          echo "desk devShell ready"
        '';
      };

      nixosConfigurations.workstation = nixpkgs.lib.nixosSystem {
        inherit system;
        modules = [ ./hosts/workstation/configuration.nix ];
      };
    };
}
```

Load the flake:

```
nix repl .#
```

Output:

```
Nix 2.35.2
Type :? for help.

Loading installable '.#'...
Added 7 variables.

nix-repl>
```

Explore outputs:

```
nix-repl> :t packages
a set

nix-repl> packages.x86_64-linux.default.name
"desk-hello"

nix-repl> :t devShells.x86_64-linux.default
a derivation

nix-repl> devShells.x86_64-linux.default.buildInputs
[ «derivation /nix/store/...-go-1.22.3.drv» ... ]
```

Explore NixOS configuration options:

```
nix-repl> nixosConfigurations.workstation.config.networking.hostName
"workstation"

nix-repl> :t nixosConfigurations.workstation.config.environment.systemPackages
a list
```

```
nix-repl> :p nixosConfigurations.workstation.options.networking.hostName.definitionsWithLocatio
[ { file = "/home/desk/infra/hosts/workstation/default.nix"; value = "workstation"; } ]
```

`.definitionsWithLocations` returns every module file and value that contributed to an option. When an unexpected service or package lands in your closure, query `.options.<path>.definitionsWithLocation` to pinpoint which nixpkgs module or import enabled it.

Build a package directly from the REPL with `:b`:

```
nix-repl> :b packages.x86_64-linux.default
```

Output:

```
This derivation produced the following outputs:
  out -> /nix/store/r7kzxq3j4b5c6d7e8f9g-desk-hello
```

After `:b` succeeds you can inspect the output path in the shell without leaving the session. `:b` is the fastest build-test loop available: no terminal switching, no retyping the flake attribute path.

After editing `flake.nix`, reload without restarting:

```
nix-repl> :r
```

Output:

```
Loading installable '.#'...
Added 7 variables.
```

Case 4: Debugging an expression with a wrong attribute path

The infrastructure desk's `shell.nix` has a typo — `pkgs.golint` instead of the correct `pkgs.golangci-lint`. Reproduce the error and then isolate it in the REPL.

Save the broken file:

```nix
# shell.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.mkShell {
  packages = [
    pkgs.go
    pkgs.golint        # wrong: this attribute was removed from nixpkgs years ago
```

```
    pkgs.gopls
  ];
}
```

Evaluate with `nix-instantiate --eval`:

```
nix-instantiate --eval shell.nix
```

Output:

```
error: attribute 'golint' missing

       at /nix/store/...-source/pkgs/top-level/all-packages.nix:9423:5:

            9422|   golang-migrate = callPackage ../tools/misc/golang-migrate { };
            9423|   golint = throw "golint was removed from nixpkgs. Use golangci-lint instead.
            9424|

       … while evaluating the attribute 'golint'
         at /home/ops/desk/shell.nix:6:5:

            5|     pkgs.go
            6|     pkgs.golint
            7|     pkgs.gopls
```

The error is an **evaluation-time** error: Nix never started building anything. Now reproduce the same inspection in the REPL to narrow it down interactively:

```
nix repl 'nixpkgs'
```

```
nix-repl> pkgs = import <nixpkgs> {}

nix-repl> :t pkgs.go
a derivation

nix-repl> :t pkgs.golangci-lint
a derivation

nix-repl> :t pkgs.golint
error: attribute 'golint' missing
       … while evaluating the attribute 'golint'
```

The REPL pinpoints exactly which path is broken. Fix the file:

```
# shell.nix
{ pkgs ? import <nixpkgs> {} }:
```

```
pkgs.mkShell {
  packages = [
    pkgs.go
    pkgs.golangci-lint   # correct attribute name
    pkgs.gopls
  ];
}
```

Re-evaluate:

```
nix-instantiate --eval shell.nix
```

Output:

```
«derivation /nix/store/...-nix-shell.drv»
```

Case 5: `builtins.trace` for printf-style debugging

When a conditional chooses the wrong branch or a version check silently produces unexpected output, `builtins.trace` prints a diagnostic line during evaluation and then returns its second argument unchanged. It is the Nix equivalent of a `print` statement.

Save as `default.nix`:

```
# default.nix
let
  pkgs = import <nixpkgs> {};

  # Confirm which version of a package is selected at eval time.
  tracedPkg = pkg:
    builtins.trace
      "checking version: ${pkg.pname}-${pkg.version}"
      pkg;

  selectedGo = tracedPkg pkgs.go;
in
{
  inherit selectedGo;
  name = selectedGo.pname;
}
```

Evaluate:

```
nix eval --file default.nix name
```

Output:

```
trace: checking version: go-1.22.3
"go"
```

The `trace:` prefix appears on `stderr`; the final value (`"go"`) appears on `stdout`. Scripts that parse `nix eval` output are unaffected by trace lines.

Chain multiple traces to track a pipeline of transformations:

```
# default.nix
let
  pkgs = import <nixpkgs> {};

  step1 = builtins.trace "step1: selecting go"    pkgs.go;
  step2 = builtins.trace "step2: selecting gopls"  pkgs.gopls;
  step3 = builtins.trace "step3: building list"    [ step1 step2 ];
in
  step3
```

Evaluate:

```
nix eval --file default.nix
```

Output:

```
trace: step1: selecting go
trace: step2: selecting gopls
trace: step3: building list
[ «derivation /nix/store/...-go-1.22.3.drv» «derivation /nix/store/...-gopls-0.15.3.drv» ]
```

Because Nix is lazy, traces only fire when the binding is actually forced. If `step2` were never referenced elsewhere, its trace would never appear. This makes `builtins.trace` a precise tool: it confirms a value was evaluated, not just defined.

Remove all `builtins.trace` calls before committing. They print to `stderr` on every evaluation, including CI and deployment pipelines, producing confusing noise in logs.

The trap

Confusing evaluation-time errors with build-time errors.

An **evaluation-time error** (`error: attribute 'foo' missing`, `error: value is a set while a string was expected`) is detected by the Nix evaluator before any builder process starts. It appears immediately in `nix eval`, `nix repl`, and `nix build` — with no build log.

A **build-time error** (a compiler error, a failing test, a missing source file) is detected by the builder process after the evaluator has successfully constructed the derivation. It appears in the build log.

Evaluation-time error — REPL:

```
nix-repl> :b packages.x86_64-linux.typo
error: flake output attribute 'packages.x86_64-linux.typo' is not a derivation or path
```

No `.drv` was written. The evaluator rejected the expression before attempting to build.

Evaluation-time error — CLI:

```
nix build .#packages.x86_64-linux.typo
```

```
error: flake output attribute 'packages.x86_64-linux.typo' is not a derivation or path
```

Build-time error — REPL:

```
nix-repl> :b packages.x86_64-linux.desk-service
building '/nix/store/abc123-desk-service.drv'...
error: builder for '/nix/store/abc123-desk-service.drv' failed with exit code 1;
       last 10 log lines:
       > ./cmd/main.go:12:2: undefined: config.LoadEnv
```

Build-time error — CLI:

```
nix build .#packages.x86_64-linux.desk-service
```

```
error: builder for '/nix/store/abc123-desk-service.drv' failed with exit code 1;
       last 10 log lines:
       > ./cmd/main.go:12:2: undefined: config.LoadEnv
```

The evaluator succeeded (a `.drv` was written). The Go compiler failed inside the sandbox. The fix belongs in the source code, not in the Nix expression.

Diagnostic rule: if you see a `/nix/store/...drv` path in the error message, the evaluator succeeded and the problem is in the build. If there is no `.drv` path, the problem is in the Nix expression itself.

The boring rule

- Open `nix repl .#` before editing any file in a flake. Keep it open alongside your editor.
- Use `:t expr` to sanity-check that a binding holds the type you expect before wiring it into a larger expression.
- Use `builtins.trace` to confirm which branch a conditional takes and which version of a package is selected. Remove all traces before committing.
- Use `:b drv` inside the REPL to test builds interactively instead of switching terminals and retyping `nix build .#...`.
- After editing `flake.nix`, type `:r` to reload the flake without restarting the session.
- A `.drv` path in the error means eval succeeded; the bug is in source code. No `.drv` means eval failed; the bug is in the Nix expression.

Try this

1. Start `nix repl 'nixpkgs'`. Find the `version` and `meta.license.shortName` of `pkgs.ripgrep`. Then use `:e pkgs.ripgrep` to open its package definition and read what `buildInputs` it declares.

2. In a flake-based project on your workstation, run `nix repl .#`. Type `:t devShells.x86_64-linux.default` to confirm it is a derivation, then `:b devShells.x86_64-linux.default` to build its environment closure. Inspect the output path with `nix path-info --closure-size /nix/store/<result>` — how large is the closure in megabytes?

3. Write a `probe.nix` that accesses a missing attribute (`pkgs.nonexistentTool`). Evaluate it with `nix-instantiate --eval probe.nix`. Then load the same file in `nix repl` with `:l probe.nix`. Does the error fire on load, or only when you type the binding name and press Enter? What does this tell you about Nix's lazy evaluation?

4. Add `builtins.trace` calls to an existing `shell.nix` or `flake.nix` to print the version of each package in a list using `builtins.map` and `builtins.concatStringsSep`. Run `nix eval --file shell.nix` and confirm the traces appear. Then remove the traces, re-evaluate, and verify `stderr` is clean.

Dev Environments

Nix Shells and the devShell Concept

Nix Shells and the devShell Concept

Compilers on the host (`apt install golang`, `brew install go`) rot the next repo. The boring default is: **`pkgs.mkShell` in the repo, entered with `nix develop` (flake) or `nix-shell` (classic), gone when you `exit`.**

Mental model

A devShell is a **child environment**, not an install:

1. Nix realises the listed packages into `/nix/store`.
2. It prepends their `bin/` to `PATH` (and sets `PKG_CONFIG_PATH`, etc.).
3. It runs `shellHook`.
4. `exit` drops every injection. The host `/usr` is unchanged.

```
host PATH = /usr/bin

       nix develop / nix-shell
          PATH = /nix/store/…-go/bin:/nix/store/…-jq/bin:…
          shellHook
       exit → host PATH again
```

File	Command
`shell.nix`	`nix-shell` (often still `<nixpkgs>`)
`flake.nix` `devShells`	`nix develop` + `flake.lock` — **desk default**

`packages` on `mkShell` is the modern list of tools. `buildInputs` / `nativeBuildInputs` still work and matter when you compile C in the shell.

Worked examples

Case 1: Classic `shell.nix`

Save as `shell.nix`:

```
# shell.nix
{ pkgs ? import <nixpkgs> { } }:
```

```
pkgs.mkShell {
  packages = [
    pkgs.ripgrep
    pkgs.jq
  ];
  shellHook = ''
    echo "=== Desk Shell Ready ==="
  '';
}
```

```
nix-shell --run 'which jq && jq --version'
```

Output (shape):

```
=== Desk Shell Ready ===
/nix/store/…-jq-…/bin/jq
jq-1.7.1
```

Outside the shell, `which jq` may be missing or a different binary.

Case 2: Declared environment

Save as `env_shell.nix`:

```
# env_shell.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.mkShell {
  packages = [ pkgs.curl ];
  env = {
    DESK_API_URL = "https://api.desk.internal";
    DESK_TIMEOUT = "30";
  };
  shellHook = ''
    echo "Targeting API: $DESK_API_URL"
  '';
}
```

```
nix-shell env_shell.nix --run 'echo Timeout: $DESK_TIMEOUT'
```

Output:

```
Targeting API: https://api.desk.internal
Timeout: 30
```

Prefer `env.` over `export` in `shellHook` so the variables are structured. Do not put secrets here.

Case 3: One-shot without a file

```
nix shell github:NixOS/nixpkgs/nixos-26.05#jq --command jq --version
```

Useful for five minutes. A repo still needs a committed shell so CI and humans match.

Case 4: Flake `devShells.default`

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [ pkgs.go pkgs.jq ];
      };
    };
}
```

```
nix develop --command go version
```

Same packages every laptop with the same `flake.lock`. Next chapter.

Case 5: `packages` vs `nativeBuildInputs`

```
pkgs.mkShell {
  packages = [ pkgs.go ];                  # on PATH
  nativeBuildInputs = [ pkgs.pkg-config ]; # also on PATH; marked build-time
  buildInputs = [ pkgs.openssl ];          # headers + libs for compiles
}
```

For a Go API with CGO off, `packages = [ pkgs.go pkgs.gopls ]` is enough. When you compile openssl-using C, you need `buildInputs`.

The trap

The trap is **`apt install golang` "so I don't need Nix for a minute."** The minute becomes the project. Two versions of `go` on `PATH`; CI uses Nix; you do not.

The other trap is secrets in `shellHook` (`export AWS_SECRET=…`). The file is git. Use sops / direnv `dotenv` that is gitignored.

The boring rule

- One `mkShell` per repo. Flake + lock when you can.
- `packages` for CLIs. `buildInputs` for libraries you link.
- `env.` for non-secret variables. `shellHook` for `echo` and `alias`.
- `exit` must restore the host. If it does not, you exported into the parent (you used `nix-shell` wrong, or direnv — part 3 later).
- No compilers via apt/brew for desk work.

Try this

1. Add `pkgs.hello` to Case 1; `nix-shell --run hello`.
2. `which jq` inside vs after `exit`.
3. `nix-shell --pure --run 'echo $PATH'` and notice `/usr/bin` is gone or last — `--pure` is how you catch host leaks.
4. Put `DESK_TOKEN` in `env` then delete it; use a gitignored `.env` instead if you needed it.

Multi-Language Development Environments

Multi-Language Development Environments

The desk API is Go, the admin UI is Node, a report is Python, and one crate talks to OpenSSL. The boring default is: **one `mkShell` listing every compiler and C library the repo actually uses — not asdf + nvm + pyenv + `libssl-dev`.**

Mental model

Version managers fight over `PATH`. Nix does not: each package is a store path. Two Pythons can exist; the shell only prepends the ones you listed.

Need	Nix
Go / Node / Python interpreters	`pkgs.go`, `pkgs.nodejs`, `python3.withPackages`
C headers / `.so`	`buildInputs` (`openssl`, `zlib`, `postgresql`)
`pkg-config`, cmake	`nativeBuildInputs`
Per-subdir tools	Named `devShells.backend` / `frontend` (direnv chapter)

Do not dump gcc+node+python into every shell "just in case." Closures grow; eval slows. Split named shells when the frontend never needs `go`.

In `mkShell`, `packages` is the PATH you type (`go`, `node`). `nativeBuildInputs` is setup-hook tools (`pkg-config`, `cmake`). `buildInputs` is libraries those hooks see (`openssl`). Mixing "everything in `packages`" skips setup hooks — `pkg-config --libs openssl` then fails even though `openssl` is "installed."

Worked examples

Case 1: One polyglot shell

Save as `polyglot.nix`:

```
# polyglot.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.mkShell {
  packages = [
```

```
    pkgs.go
    pkgs.nodejs
    (pkgs.python3.withPackages (ps: [ ps.requests ps.pydantic ]))
    pkgs.sqlite
  ];
  shellHook = ''
    echo "Polyglot: $(go env GOVERSION) node $(node -v) $(python3 --version)"
  '';
}
```

```
nix-shell polyglot.nix --run 'go env GOVERSION; node -v; python3 --version'
```

Output (shape — versions follow 26.05):

```
go1.24.x
v22.x.x
Python 3.12.x
```

Use `pkgs.nodejs` (channel default on 26.05), not a random `nodejs_22` unless you have a reason to pin a major. `pkgs.go` is the channel Go — `buildGoModule` in CI must use the **same** major or `go.mod`'s `toolchain` line fights `GOTOOLCHAIN=local`.

```
nix-shell polyglot.nix --run 'which go; which node; which python3'
```

All three must be `/nix/store/…`. A mix of `/usr/bin/python3` and store `go` is two version managers again. `PATH` order in the shell is Nix first.

Case 2: OpenSSL via pkg-config

Save as `native_c.nix`:

```
# native_c.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.mkShell {
  nativeBuildInputs = [ pkgs.pkg-config ];
  buildInputs = [ pkgs.openssl pkgs.zlib ];
  shellHook = ''
    pkg-config --cflags openssl
  '';
}
```

```
nix-shell native_c.nix --run 'pkg-config --libs openssl'
```

Output (shape):

`-L/nix/store/…-openssl-…/lib -lssl -lcrypto`

Headers are **not** `/usr/include/openssl`. `PKG_CONFIG_PATH` is set by the setup hook because `pkg-config` is in `nativeBuildInputs`. Put `openssl` only in `packages` and the hook never runs.

Case 3: Named shells in a flake

```
# flake.nix fragment
devShells.x86_64-linux = {
  default = self.devShells.x86_64-linux.backend;
  backend = pkgs.mkShell { packages = [ pkgs.go pkgs.gopls ]; };
  frontend = pkgs.mkShell { packages = [ pkgs.nodejs pkgs.pnpm ]; };
};
```

```
nix develop .#frontend --command which node
nix develop .#backend --command which go
```

direnv: `use flake .#frontend` in `web/.envrc`.

Case 4: Python packages vs a second venv

```
packages = [
  (pkgs.python3.withPackages (ps: [ ps.requests ]))
];
```

`python3 -c "import requests"` works. `pip install requests` inside this shell writes to a user venv and fights Nix. If you need a lockfile-heavy app, package it (`buildPythonApplication`) instead of pip in the shell.

Case 5: `LD_LIBRARY_PATH` is a last resort

Nix wrappers set RPATH. If a **vendor** binary still cannot find `libssl.so`, use `nix-ld` / FHS (later chapter), not:

```
export LD_LIBRARY_PATH=$(nix eval --raw nixpkgs#openssl.out)/lib
```

That leaks into every child process.

The trap

The trap is **`buildInputs = [ pkgs.go pkgs.nodejs pkgs.python3 pkgs.gcc pkgs.cmake pkgs.openssl … ] copied from another team.`** The shell takes a minute to eval and nobody uses cmake. List what `README` says you run.

The other trap is host `/usr/include` for openssl while Node's `node-gyp` uses Nix python. Mix = "works on Ubuntu, dies on macOS."

A third: `packages = [ pkgs.openssl pkgs.pkg-config ]` and wondering why `pkg-config` sees nothing — use `nativeBuildInputs` / `buildInputs`. A fourth: `pip install` / `npm i -g` inside `nix develop` "just this once."

The boring rule

- One shell per *job* (backend / frontend), not one kitchen sink.
- Interpreters in `packages`. C libs in `buildInputs`. cmake/pkg-config in `nativeBuildInputs`.
- `python3.withPackages` instead of pip. Same Go major as `buildGoModule`.
- Pin majors (`nodejs_22`) only when the default is wrong.
- No host headers. No global `LD_LIBRARY_PATH`.

Try this

1. Add `pkgs.cargo` to Case 1; `nix-shell --run 'cargo --version'`.
2. `pkg-config --cflags openssl` in Case 2; confirm `/nix/store`.
3. Split Case 3; `which node` must fail in `.#backend` if node is not listed.
4. `nix-shell --pure --run 'python3 -c "import ssl; print(ssl.OPENSSL_VERSION)"'` with openssl in `buildInputs`.
5. Move `pkg-config` from `packages` to `nativeBuildInputs` and confirm `pkg-config --libs openssl` starts working.

Flakes for Reproducible Dev Environments

Flakes for Reproducible Dev Environments

`shell.nix` plus `<nixpkgs>` is "whatever I last `nix-channel --update`'d." The boring default is: **`flake.nix` + committed `flake.lock` on `nixos-26.05`, `devShells.${system}.default`, `nix develop`**.

Mental model

Piece	Job
`inputs.nixpkgs.url`	Which train (`nixos-26.05`)
`flake.lock`	Exact `rev` + `narHash`
`devShells.${system}.default`	What `nix develop` enters
Pure eval	No undeclared files, no ambient `NIX_PATH`

`flake.nix → flake.lock → identical mkShell on every machine`

`nix flake update` is a deliberate PR. Not a side effect of `nix develop`.

Worked examples

Case 1: Minimal flake shell

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk service development shell";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      devShells.${system}.default = pkgs.mkShell {
```

```
        packages = [ pkgs.go pkgs.gopls pkgs.golangci-lint ];
        env.GOTOOLCHAIN = "local";
      };
    };
}
```

```
nix flake lock
nix develop --command go env GOVERSION GOTOOLCHAIN
```

Output (shape):

```
go1.24.x
local
```

Apple silicon: `system = "aarch64-darwin";` or `eachDefaultSystem` (Case 3).

Case 2: Commit the lock

```
git add flake.nix flake.lock
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.locked.rev'
```

Without `flake.lock` in git, CI generates a new one every job. That is a channel with extra steps.

```
git check-ignore -v flake.lock || echo 'lock is tracked (good)'
```

If `.gitignore` has `flake.lock`, delete that line. The lock is the pin.

```
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.original.ref // .locks.nodes.nixpkgs
```

You want `nixos-26.05` (or a commit on that train), not `nixpkgs-unstable`.

Case 3: `eachDefaultSystem`

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }:
    let
      systems = [ "x86_64-linux" "aarch64-linux" "aarch64-darwin" ];
      forAll = nixpkgs.lib.genAttrs systems;
    in
    {
      devShells = forAll (system:
        let pkgs = nixpkgs.legacyPackages.${system}; in {
```

```
          default = pkgs.mkShell { packages = [ pkgs.go pkgs.jq ]; };
        });
    };
}
```

`flake-utils` is optional sugar. Explicit `systems` is easier to read.

Case 4: Update one input

```
nix flake update nixpkgs
git diff flake.lock
nix develop --command go env GOVERSION
```

Read the diff. If gcc moved, CI will compile. That is expected. Do not `update` in the same PR as a feature.

Case 5: `nix flake check` as the empty gate

```
checks.x86_64-linux.go-version = pkgs.runCommand "go-version" {
  nativeBuildInputs = [ pkgs.go ];
} ''
  go env GOVERSION > $out
'';
```

```
nix flake check
```

Later chapters add real tests. Even this proves CI uses the lock.

The trap

The trap is **gitignore `flake.lock` "because it churns."** Then there is no team pin.

The other trap is `inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";` for a production service. Unstable moves daily. 26.05 until you have an upgrade PR.

`nix develop --impure` to read `$HOME` is a smell. Declare the file as an input or keep it out of eval.

The boring rule

- `nixos-26.05` + `flake.lock` in git.
- `nix develop` for the default shell.
- `genAttrs` / explicit systems; do not hardcode only `x86_64-linux` if the desk uses Macs.

- `nix flake update nixpkgs` in its own commit.
- `GOTOOLCHAIN=local` in Go shells so the Nix `go` wins.

Try this

1. `jq -r '.nodes.nixpkgs.locked.rev' flake.lock`.
2. Delete `flake.lock` locally, `nix develop`, compare the new rev, restore from git.
3. Add `aarch64-linux` to `systems`; `nix flake show`.
4. `nix flake update --commit-lock-file` and read the commit (then reset if you should not keep it).

Environment Management with Home Manager

Environment Management with Home Manager

`nix develop` is the **project**. Home Manager is the **person**. The boring default is: **personal git/prompt/editor in Home Manager; compilers only in the project's devShell — never Go 1.x in `home.packages` "for every repo"**.

Part 7 teaches Home Manager as a NixOS user module. This chapter is the boundary with part 3's shells, including on a foreign Linux laptop (Nix 2.35, nixpkgs 26.05).

Mental model

```
~/.nix-profile     Home Manager: jq, bat, git config, nvim
direnv / nix develop
    PATH prepends  go, gopls, postgresql client  for this repo only
```

When you `cd` out, the compiler vanishes. Your `git` aliases stay. That split is how two repos can pin two Go versions without fighting.

`home.stateVersion = "26.05";` on a **new** user born on 26.05. If the user was created on 24.11, leave `"24.11"`. Birth, not channel.

On NixOS 26.05, Home Manager `follows` the system nixpkgs. On Ubuntu + Nix 2.35, pin `nixos-26.05` in the **user** flake too.

Worked examples

Case 1: Personal tools only

Save as `home.nix`:

```nix
# home.nix
{ pkgs, ... }:

{
  home.username = "deskuser";
  home.homeDirectory = "/home/deskuser";
  home.stateVersion = "26.05";

  home.packages = with pkgs; [
    htop
```

```
    bat
    fzf
    jq
    ripgrep
  ];

  programs.git = {
    enable = true;
    userName = "Desk Engineer";
    userEmail = "engineer@desk.corp";
  };
}
```

```
home-manager switch --flake .#deskuser
which jq
```

`jq` is a store path. It is not `gcc`. `which go` after this switch must still fail (or be distro go you should not use).

Case 2: Project compiler stays in the flake

Save as `flake.nix` in the service repo:

```
# flake.nix
{
  description = "Desk API";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      devShells.${system}.default = pkgs.mkShell {
        packages = [ pkgs.go pkgs.gopls pkgs.postgresql ];
      };
    };
}
```

```
# outside the shell
which go || echo 'no go'
nix develop --command which go
```

Home Manager must **not** also install `pkgs.go`. If it does, `which go` outside the project still works — and CI (which only has the flake) does not match your laptop.

Case 3: direnv composes, it does not replace HM

Save as `.envrc`:

```
# .envrc
use flake
```

```
direnv allow
echo "$PATH" | tr ':' '\n' | head
```

`use flake` prepends the devShell. Starship from Home Manager still runs. You do not put `programs.git` in the project flake.

Case 4: Same nixpkgs as the OS when you can

On NixOS, in the **user** flake:

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  inputs.home-manager.url = "github:nix-community/home-manager/release-26.05";
  inputs.home-manager.inputs.nixpkgs.follows = "nixpkgs";
}
```

```
nix flake metadata
```

One nixpkgs. HM `release-24.11` + nixpkgs 26.05 is eval pain. On Ubuntu + Nix 2.35, the same pin keeps `jq` the same closure as CI.

Case 5: Rollback is per layer

```
home-manager generations
home-manager switch --rollback
```

This does not roll back the project's `go`. `nix develop` is not a Home Manager generation. Two rollbacks, two tools, two meanings. A bad `home.packages` does not warrant `nixos-rebuild switch --rollback`.

The trap

The trap is `home.packages = [ pkgs.go pkgs.nodejs pkgs.rustc ];` as a "developer workstation." The next repo needs another version. You overlay. CI diverges. Put toolchains in the repo flake.

The other trap is bumping `home.stateVersion` to `"26.05"` on a user born on 24.11 because "we upgraded the channel."

The boring rule

- Home Manager: identity, editor, prompt, generic CLIs.
- devShell: language toolchains and DB clients the app needs.
- `stateVersion` is birth, not "current channel."
- Same nixpkgs pin as the desk OS when the machine is NixOS 26.05. HM `release-26.05` + `follows`.
- Do not install compilers globally "so PATH always has go."

Try this

1. Add `pkgs.hello` to `home.packages`, switch, `hello`, remove it, switch.
2. `which go` in and out of `nix develop` on a repo that lists `pkgs.go`. Out must fail (or be distro go you should not use).
3. `echo $PATH` inside direnv and label HM vs devShell entries.
4. Set `home.stateVersion` wrongly to a future value on a lab user, read the warning, restore the birth value.

Migrating from Docker Desktop to Nix DevShells

Migrating from Docker Desktop to Nix DevShells

`docker run golang:1.24 go test` is a VM, a daemon, and a bind mount to run a compiler. The boring default is: **Nix `devShell` for `go` / `gopls` / linters; containers only for Postgres, Redis, and other long-running state**.

Mental model

	Docker Desktop for `go test`	Nix `nix develop --command go test`
What runs	Linux VM + dockerd + container	Host `go` from `/nix/store`
Startup	Seconds to tens of seconds	Milliseconds after first substitute
Editor	Dev Containers / remote Linux	Local `gopls` on PATH
Files	Virtio/osxfs bind mounts	Native disk

Keep Docker/Podman for **databases**. Do not keep it for **compilers**. Same `pkgs.go` as `buildGoModule` on 26.05; `GOTOOLCHAIN=local` so Go does not download another SDK.

Worked examples

Case 1: Replace a compile container

Was:

```
docker run --rm -v "$PWD":/src -w /src golang:1.24 go test ./...
```

Save as `flake.nix`:

```
# flake.nix
{
  description = "Native desk Go environment";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
```

```
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [ pkgs.go pkgs.gopls pkgs.golangci-lint pkgs.delve ];
        env.GOTOOLCHAIN = "local";
        env.CGO_ENABLED = "0";
      };
    };
}
```

```
nix develop --command go test ./...
```

Output (shape):

```
ok      desk/service    0.015s
```

Same tests, no VM.

Case 2: Time it once

```
time nix develop --command go version
time docker run --rm golang:1.24 go version
```

After Nix has substituted `go`, the first command should be far cheaper. Docker always pays daemon + create + start. Write the numbers in a comment in the flake so the next argument dies.

Case 3: Hybrid — Nix CLI, Podman Postgres

Save as `start_db.sh`:

```
# start_db.sh
#!/usr/bin/env bash
set -euo pipefail
podman rm -f desk-db >/dev/null 2>&1 || true
podman run -d --name desk-db \
  -p 5432:5432 \
  -e POSTGRES_PASSWORD=desk-dev \
  postgres:16-alpine
```

Save as the shell packages line:

```
# flake.nix fragment
{
  packages = [ pkgs.go pkgs.postgresql pkgs.podman ];
```

```
}
```

```
nix develop --command bash start_db.sh
nix develop --command psql postgres://postgres:desk-dev@127.0.0.1:5432/postgres -c 'SELECT 1
```

The **server** is a container. The **client** (`psql`, `go`) is Nix. Password is a **dev** secret; production uses sops.

Case 4: Editor stays on the host

```
nix develop --command which gopls
```

VS Code / Zed / Neovim should see that path via direnv (`use flake`). Disable Dev Containers for this repo. If `gopls` is missing, add `pkgs.gopls` to the shell, not a `golang` image tag.

Case 5: When to keep Compose

Keep `compose.yml` for:

- Postgres + Redis + localstack **as a set**
- Matching production's Linux userspace for a **weird** native addon

Do not keep `golang` / `node` / `rust` services in Compose if Nix already provides them.

```
# compose.yml - keep
services:
  db:
    image: postgres:16-alpine
    ports: ["5432:5432"]
# delete:
#  go:
#    image: golang:1.24
```

CI: `nix develop --command go test` (or `buildGoModule doCheck`), not `docker compose run go`.

The trap

The trap is **moving the IDE into the container** because "that is how we did Node." You wanted PATH isolation. Nix already did that. Remote-container then fights file watchers and extensions.

The other trap is pinning `golang:latest` in CI and Nix `go` on the laptop. Same pin: 26.05 `pkgs.go` in both, or `buildGoModule` in CI and no golang image.

The boring rule

- Compilers and LSPs: Nix devShell.
- Stateful daemons: Podman/Docker.
- Time `go version` both ways once; keep the faster one for the compiler.
- `GOTOOLCHAIN=local`. Same `go` as `buildGoModule`.
- Dev Compose passwords stay out of production flakes.

Try this

1. Case 2 timings; write the numbers in a comment in the flake.
2. `which gopls` after `direnv allow`.
3. Start Postgres with Case 3; `go test` against `localhost:5432` without a `golang` container.
4. Delete the `golang:` service from Compose if it only ran tests.

Automatic Dev Shells with direnv

Automatic Dev Shells with direnv

Running `nix develop` by hand is ceremony you will forget. `direnv` eliminates it: the moment you `cd` into a project directory, your shell gains every tool that project declared. `nix-direnv` makes that transition instant and GC-safe. Together they are the boring default for any infrastructure desk running multiple projects side by side.

Mental model

`direnv` is a shell hook. After you add it to your shell's init file, it intercepts every directory change. When the new directory contains a file named `.envrc`, direnv evaluates that file and exports the resulting environment variables into your running shell. When you leave the directory, those variables vanish.

`nix-direnv` provides a single function — `use flake` — that you call from `.envrc`. It replaces direnv's slower built-in `use nix` with a version that:

1. Builds the devShell once and caches the result in `/nix/store`.
2. Creates a GC-root symlink at `.direnv/flake-profile` so that `nix-collect-garbage -d` cannot evict the store paths the shell depends on.
3. Re-evaluates only when `flake.nix`, `flake.lock`, or `.envrc` changes — not on every `cd`.

Lifecycle

```
cd project/
    direnv detects .envrc
         nix-direnv: cache hit? → restore env (ms)
                       cache miss? → nix build → cache → restore env (s/min, once)
              PATH, PKG_CONFIG_PATH, … exported into current shell
cd ..
    direnv unloads the environment
```

The only files a project needs beyond `flake.nix` are:

File	Purpose	Commit?
`.envrc`	tells direnv to `use flake`	**yes**
`.direnv/`	nix-direnv cache and GC root	**no** (gitignore)

Worked examples

Case 1: Installing direnv and nix-direnv via Home Manager

Home Manager is the standard way to install direnv on a NixOS workstation. The module handles shell hook injection automatically — you do not touch `.bashrc` or `.zshrc` by hand.

Save as `home.nix`:

```
# home.nix
{ pkgs, ... }:

{
  programs.direnv = {
    enable = true;            # installs direnv and adds the shell hook
    nix-direnv.enable = true; # replaces built-in use_nix with nix-direnv
  };

  # Optional: silence direnv on every cd; it still prints on changes.
  home.sessionVariables = {
    DIRENV_LOG_FORMAT = "";
  };
}
```

Apply the configuration:

```
home-manager switch
```

Output:

```
Starting Home Manager activation
Activating checkFilesChanged
Activating installPackages
replacing /home/ops/.nix-profile → /nix/store/…-home-manager-path
Activating linkGeneration
Activating onFilesChange
Activating reloadSystemd
```

Open a new shell (or `exec $SHELL`) so the hook takes effect. Now trust the first project directory:

```
direnv allow ~/projects/desk
```

Output:

```
direnv: loading ~/projects/desk/.envrc
direnv: using flake
direnv: nix-direnv: renewed cache
direnv: export +AR +AS +CC +CONFIG_SHELL … +buildInputs
```

`direnv allow` is a one-time command per directory. It records a hash of `.envrc` in `~/.local/share/direnv/allow/`. If `.envrc` changes later you must run `direnv allow` again — a deliberate safety gate.

Case 2: A Go desk project

The infrastructure desk maintains a small Go service that talks to the internal metrics API. The flake declares a devShell with `go`, `gopls`, and `gotools`. The `.envrc` is a single line.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk metrics service";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages.${system};
      in
      {
        devShells.default = pkgs.mkShell {
          name = "desk-metrics";
          packages = [
            pkgs.go_1_23
            pkgs.gopls
            pkgs.gotools  # goimports, etc.
          ];

          shellHook = ''
            echo "desk-metrics shell: $(go version)"
          '';
        };
      });
}
```

Save as `.envrc`:

```
# .envrc
use flake
watch_file flake.lock
# optional: local secrets not in git
```

```
# dotenv_if_exists .envrc.local
```

Allow and enter:

```
cd ~/projects/desk-metrics
direnv allow
```

Output:

```
direnv: loading ~/projects/desk-metrics/.envrc
direnv: using flake
direnv: nix-direnv: building flake output 'devShell.x86_64-linux.default'
[… nix build output …]
direnv: nix-direnv: renewed cache
desk-metrics shell: go version go1.23.4 linux/amd64
direnv: export +AR +AS +CC +CGO_ENABLED +GOPATH … +buildInputs
```

Subsequent entries are instant:

```
cd ..
cd ~/projects/desk-metrics
```

Output:

```
direnv: loading ~/projects/desk-metrics/.envrc
direnv: using flake
direnv: nix-direnv: using cached dev shell
desk-metrics shell: go version go1.23.4 linux/amd64
direnv: export +AR +AS +CC +CGO_ENABLED +GOPATH … +buildInputs
```

Verify the toolchain:

```
which go gopls goimports
```

Output:

```
/nix/store/…-go-1.23.4/bin/go
/nix/store/…-gopls-0.16.2/bin/gopls
/nix/store/…-gotools-2024-01-01/bin/goimports
```

Case 3: A polyglot project with named devShells

The desk's internal portal has a Go backend and a Node/TypeScript frontend. They live in the same repository but need separate toolchains. A single flake with two named devShells keeps everything in one place.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk portal - backend + frontend";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    flake-utils.url = "github:numtide/flake-utils";
  };

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        pkgs = nixpkgs.legacyPackages.${system};
      in
      {
        devShells.backend = pkgs.mkShell {
          name = "portal-backend";
          packages = [
            pkgs.go_1_23
            pkgs.gopls
            pkgs.gotools
            pkgs.sqlite
          ];
          shellHook = ''
            echo "[backend] $(go version)"
          '';
        };

        devShells.frontend = pkgs.mkShell {
          name = "portal-frontend";
          packages = [
            pkgs.nodejs_22
            pkgs.nodePackages.typescript
            pkgs.nodePackages.typescript-language-server
          ];
          shellHook = ''
            echo "[frontend] node $(node --version)"
          '';
        };
      });
}
```

Each subdirectory gets its own `.envrc` pointing at the correct output.

Save as `backend/.envrc`:

```
# backend/.envrc
use flake ..#backend
```

Save as `frontend/.envrc`:

```
# frontend/.envrc
use flake ..#frontend
```

Allow both, then switch between them:

```
direnv allow backend
direnv allow frontend
cd backend
```

Output:

```
direnv: loading ~/projects/portal/backend/.envrc
direnv: using flake ..#backend
direnv: nix-direnv: using cached dev shell
[backend] go version go1.23.4 linux/amd64
direnv: export +AR +AS +CC … +buildInputs
```

```
cd ../frontend
```

Output:

```
direnv: unloading
direnv: loading ~/projects/portal/frontend/.envrc
direnv: using flake ..#frontend
direnv: nix-direnv: using cached dev shell
[frontend] node v22.9.0
direnv: export +NODE_PATH … +buildInputs
```

The Go tools are absent from the frontend shell and vice versa. No manual activation or deactivation is needed.

Case 4: Caching and GC safety

Every time nix-direnv builds a devShell it writes a symlink under `.direnv/` that acts as a Nix GC root. This prevents `nix-collect-garbage -d` from deleting the store paths your shell depends on.

Inspect the structure after the first `use flake`:

```
ls -la ~/projects/desk-metrics/.direnv/
```

Output:

```
total 16
drwxr-xr-x 3 ops ops 4096 Sep  9 10:12 .
drwxr-xr-x 8 ops ops 4096 Sep  9 10:12 ..
drwxr-xr-x 2 ops ops 4096 Sep  9 10:12 flake-inputs
lrwxrwxrwx 1 ops ops   80 Sep  9 10:12 flake-profile -> /nix/store/…-desk-metrics-profile
lrwxrwxrwx 1 ops ops   80 Sep  9 10:12 flake-profile-1-link -> /nix/store/…-desk-metrics-profil
```

```
ls -la ~/projects/desk-metrics/.direnv/flake-inputs/
```

Output:

```
total 8
drwxr-xr-x 2 ops ops 4096 Sep  9 10:12 .
drwxr-xr-x 3 ops ops 4096 Sep  9 10:12 ..
lrwxrwxrwx 1 ops ops   83 Sep  9 10:12 nixpkgs -> /nix/store/…-source
```

Now run garbage collection:

```
nix-collect-garbage -d
```

Output:

```
finding garbage collector roots...
removing stale link from '/nix/var/nix/gcroots/per-user/ops/…' to '…'
deleting garbage...
deleting '/nix/store/…-some-old-package'
…
2.31 GiB freed
```

Your devShell store paths survive because `.direnv/flake-profile-1-link` is registered as a GC root. Re-enter the directory:

```
cd ~/projects/desk-metrics
```

Output:

```
direnv: loading ~/projects/desk-metrics/.envrc
direnv: using flake
direnv: nix-direnv: using cached dev shell
desk-metrics shell: go version go1.23.4 linux/amd64
direnv: export +AR …
```

The shell is still instant — nothing was collected.

Add `.direnv/` to the project's `.gitignore`:

```
# .gitignore
.direnv/
```

The symlinks are machine-local. Committing them would break other developers' paths.

The trap

Using plain `use nix` instead of `use flake`

direnv ships with a built-in `use_nix` helper for legacy `shell.nix` files. A common mistake is writing `.envrc` like this for a flake project:

```
# .envrc  ← WRONG for a flake project
use nix
```

`use nix` has no cache. It invokes `nix-shell` on every directory entry. For a devShell with many dependencies, that means several seconds of evaluation on every `cd`.

Symptom:

```
cd ~/projects/desk-metrics
```

Output:

```
direnv: loading ~/projects/desk-metrics/.envrc
direnv: using nix
[4.2s pause]
direnv: export +AR +AS +CC …
```

Fix: use `use flake` (provided by nix-direnv):

```
# .envrc  ← correct
use flake
```

Result:

```
cd ~/projects/desk-metrics
```

Output:

```
direnv: loading ~/projects/desk-metrics/.envrc
direnv: using flake
direnv: nix-direnv: using cached dev shell
direnv: export +AR +AS +CC …
```

The difference is a cache layer. `use flake` stores the built environment as a profile and reloads it by reading symlinks — no Nix evaluation occurs on a cache hit.

A related mistake is forgetting to re-run `direnv allow` after editing `.envrc`. direnv blocks evaluation of changed files until you explicitly approve them:

```
direnv: error ~/projects/desk-metrics/.envrc is blocked. Run `direnv allow` to approve its cont
```

This is a security feature, not a bug. Run `direnv allow` after every intentional `.envrc` change.

The boring rule

- **Commit `.envrc`** with `use flake` (or `use flake .#<name>`) plus `watch_file flake.lock`. A lock bump must rebuild the cache. `.envrc.local` / `.env` stay gitignored (`dotenv_if_exists`).
- **Gitignore `.direnv/`**. Its contents are machine-local GC roots and cached profiles.
- **Run `direnv allow` explicitly** every time `.envrc` changes. Treat it as a checkpoint: you reviewed the file and approved it.
- **Never use `use nix` in a flake project.** The built-in has no cache and no GC safety. Always use the nix-direnv `use flake`.
- **Install via Home Manager**, not ad-hoc. The `programs.direnv` module wires the shell hook correctly for bash, zsh, and fish without manual init-file edits.
- **One `.envrc` per shell boundary.** Subdirectory `.envrc` files can reference the parent flake with `use flake ..#<name>`. Do not duplicate flake outputs.

Try this

1. **Baseline timing.** In an existing flake project, write an `.envrc` with `use nix` and measure the cold entry time with `time (cd project && cd ..)`. Then switch to `use flake`, run `direnv allow`, and measure again. Record both numbers and note the difference.
2. **Named-shell workspace.** Create a `~/projects/portal` repository with the two-shell flake from Case 3. Add `backend/.envrc` and `frontend/.envrc`. Verify that `which go` returns a path only inside `backend/` and `which node` returns a path only inside `frontend/`.
3. **GC resilience.** Enter a direnv-managed project and note a store path from `echo $buildInputs | tr ' ' '\n' | head -1`. Run `nix-collect-garbage -d`. Confirm the store path still exists with `ls /nix/store/<path>`. Then delete `.direnv/` manually, re-enter the directory, and observe nix-direnv rebuilding from scratch.
4. **Audit your trust.** Run `cat ~/.local/share/direnv/allow/*` to list all allowed `.envrc` hashes on your workstation. For each corresponding directory, open `.envrc` and confirm the content matches what you expect. Revoke any stale entries with `direnv deny <path>`.

nix-darwin: Declarative macOS Configuration

nix-darwin: Declarative macOS Configuration

Half the infrastructure desk runs macOS laptops; the other half runs NixOS servers. Without `nix-darwin`, the macOS machines drift — Homebrew packages installed by hand, `defaults write` commands nobody remembers, and a setup doc that is always three months out of date. The boring default is: **declare macOS system configuration in the same flake that manages Linux, and apply it with `darwin-rebuild switch`**.

Mental model

NixOS uses `nixosConfigurations` to describe Linux machines. `nix-darwin` adds `darwinConfigurations` for macOS machines. Both live in the same `flake.nix`.

```
flake.nix
    nixosConfigurations.desk-server   ← nixos-rebuild switch
    darwinConfigurations.desk-mac     ← darwin-rebuild switch
```

`nix-darwin` manages:

Layer	Managed by nix-darwin
System packages	`environment.systemPackages`
Homebrew casks	`homebrew.casks` (via `nix-homebrew`)
launchd daemons & agents	`launchd.daemons`, `launchd.agents`
macOS defaults	`system.defaults.*`
Environment variables	`environment.variables`
Shell initialisation	`programs.zsh`, `programs.fish`
Per-user config	Home Manager as a module

`nix-darwin` **cannot** manage SIP-protected system paths, kernel extensions, or anything that macOS's System Integrity Protection locks away. Attempting to do so produces a clear error at build time — not a silent failure at runtime.

The apply command is the macOS twin of `nixos-rebuild switch`:

```
nixos-rebuild switch --flake .#desk-server   ← Linux
sudo darwin-rebuild switch --flake .#desk-mac     ← macOS
```

Worked examples

Case 1: Minimal darwinConfiguration

This is the smallest working `nix-darwin` flake. It sets two `system.defaults` tweaks that every developer on the team wants: auto-hiding Dock and a fast key-repeat rate.

`nix-darwin` is not part of `nixpkgs`. Pin `nix-darwin-26.05`, not `master`. A Mac-only flake may use `nixpkgs-26.05-darwin` (Hydra darwin binaries). A mixed Linux+Mac flake may `follows` `nixos-26.05` — some darwin attrs then build from source.

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-mac - minimal nix-darwin configuration";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nix-darwin = {
      url = "github:nix-darwin/nix-darwin/nix-darwin-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nix-darwin }:
    let
      system = "aarch64-darwin";           # M-series Mac; use x86_64-darwin for Intel
      pkgs   = nixpkgs.legacyPackages.${system};
    in
    {
      darwinConfigurations."desk-mac" = nix-darwin.lib.darwinSystem {
        inherit system;
        modules = [
          ({ pkgs, ... }: {
            # System packages available to all users
            environment.systemPackages = [
              pkgs.ripgrep
              pkgs.jq
              pkgs.git
            ];

            # macOS system defaults
            system.defaults = {
              dock.autohide = true;
              dock.autohide-delay = 0.0;
              dock.autohide-time-modifier = 0.2;
```

```
            NSGlobalDomain.InitialKeyRepeat = 15;   # delay before repeat (lower = faster)
            NSGlobalDomain.KeyRepeat = 2;            # repeat interval (lower = faster)
          };

          # Integer birth, not a channel string. New 26.05 machines: 6.
          # Do not bump to 7 without reading `darwin-rebuild changelog`.
          system.stateVersion = 6;

          # Activation runs as root. Defaults/homebrew apply to this user.
          system.primaryUser = "deskadmin";

          # 26.05 channel Nix is 2.34. If the installer already put 2.35+
          # on the Mac, omit nix.package so you do not downgrade the daemon.
          # nix.package = pkgs.nix;

          nixpkgs.config.allowUnfree = true;
        })
      ];
    };
  };
}
```

Install `nix-darwin` once on a fresh Mac, then apply as **root** (activation is root-only):

```
sudo nix run nix-darwin/nix-darwin-26.05#darwin-rebuild -- switch --flake .#desk-mac
# after the first generation:
sudo sudo darwin-rebuild switch --flake .#desk-mac
```

Output:

```
building the system configuration...
setting up /etc/zshrc...
setting up /etc/bashrc...
setting up /etc/nix/nix.conf...
system defaults applied: dock.autohide = true, KeyRepeat = 2
reloading Dock...
setting up launchd services...
/run/current-system -> /nix/store/3zqr9m4s...-darwin-system

System generation: desk-mac-1  (2024-11-20)
```

Every subsequent `darwin-rebuild switch` is idempotent. Running it twice produces no changes.

Case 2: One flake for Linux and macOS

The desk team maintains one repository. `desk-server` is a NixOS machine. `desk-mac` is a developer's MacBook. Both receive the same core tools from a shared `commonPackages` list.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk infrastructure - Linux + macOS from one flake";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nix-darwin = {
      url = "github:nix-darwin/nix-darwin/nix-darwin-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nix-darwin }:
    let
      # Packages every desk engineer needs regardless of OS
      commonPackages = pkgs: [
        pkgs.ripgrep
        pkgs.jq
        pkgs.git
        pkgs.go
      ];
    in
    {
      #   Linux server
      nixosConfigurations."desk-server" = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ({ pkgs, ... }: {
            environment.systemPackages = commonPackages pkgs;

            # Server-only: enable the SSH daemon
            services.openssh.enable = true;

            system.stateVersion = "26.05";
          })
        ];
      };

      #   macOS workstation
      darwinConfigurations."desk-mac" = nix-darwin.lib.darwinSystem {
        system = "aarch64-darwin";
        modules = [
          ({ pkgs, ... }: {
            environment.systemPackages = commonPackages pkgs ++ [
```

```
            pkgs.mas   # Mac App Store CLI - macOS only
          ];

          system.defaults.dock.autohide = true;

          system.stateVersion = 6;
          system.primaryUser = "deskadmin";
          nixpkgs.config.allowUnfree = true;
        })
      ];
    };
  };
}
```

Apply to the server (run on the Linux machine):

```
nixos-rebuild switch --flake .#desk-server
```

Output:

```
building the system configuration...
activating the configuration...
setting up /etc/ssh/sshd_config...
starting sshd.service
```

Apply to the Mac (run on the macOS machine):

```
sudo darwin-rebuild switch --flake .#desk-mac
```

Output:

```
building the system configuration...
system defaults applied
setting up /etc/zshrc...
/run/current-system -> /nix/store/7f2kp1rs...-darwin-system
```

`ripgrep`, `jq`, `git`, and `go` are now in `/run/current-system/sw/bin` on both machines. Both were updated from a single source of truth.

Case 3: Declarative Homebrew casks via nix-homebrew

`nix-darwin` can drive Homebrew declaratively through the `nix-homebrew` module. GUI macOS apps — which are not in `nixpkgs` — are listed in code and installed automatically during `darwin-rebuild switch`. No manual `brew install --cask` commands.

`nix-homebrew` is a separate flake input that bridges `nix-darwin` and Homebrew. It installs Homebrew itself under `/opt/homebrew` if absent and keeps the tap and cask list in sync.

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-mac with declarative Homebrew casks";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nix-darwin = {
      url = "github:nix-darwin/nix-darwin/nix-darwin-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    nix-homebrew = {
      url = "github:zhaofengli/nix-homebrew";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    homebrew-core = {
      url = "github:homebrew/homebrew-core";
      flake = false;
    };

    homebrew-cask = {
      url = "github:homebrew/homebrew-cask";
      flake = false;
    };
  };

  outputs = { self, nixpkgs, nix-darwin, nix-homebrew,
              homebrew-core, homebrew-cask }:
  {
    darwinConfigurations."desk-mac" = nix-darwin.lib.darwinSystem {
      system = "aarch64-darwin";
      modules = [
        # Wire in nix-homebrew
        nix-homebrew.darwinModules.nix-homebrew
        {
          nix-homebrew = {
            enable = true;
            enableRosetta = true;    # required on Apple Silicon for x86_64 casks
            user = "alice";          # the user that owns the Homebrew prefix

            taps = {
              "homebrew/homebrew-core" = homebrew-core;
              "homebrew/homebrew-cask" = homebrew-cask;
```

```
            };

            # Prevent Homebrew from auto-updating outside nix-darwin
            autoMigrate = true;
          };
        }

        ({ pkgs, ... }: {
          environment.systemPackages = [
            pkgs.ripgrep
            pkgs.jq
            pkgs.git
          ];

          # Casks are GUI apps not available in nixpkgs
          homebrew = {
            enable = true;

            casks = [
              "ghostty"     # GPU-accelerated terminal
              "1password"   # password manager
              "arc"         # browser
            ];

            # Remove casks that are no longer in the list on next switch
            onActivation.cleanup = "zap";
          };

          system.stateVersion = 6;
          system.primaryUser = "deskadmin";
          nixpkgs.config.allowUnfree = true;
        })
      ];
    };
  };
}
```

Apply the configuration:

```
sudo darwin-rebuild switch --flake .#desk-mac
```

Output:

```
building the system configuration...
==> Homebrew taps are up to date.
==> Installing cask ghostty
==> Installing cask 1password
==> Installing cask arc
```

```
system defaults applied
/run/current-system -> /nix/store/9xbp3v7c...-darwin-system
```

Adding a cask is a one-line diff in `homebrew.casks`. Removing it from the list and running `darwin-rebuild switch` uninstalls it automatically because `onActivation.cleanup = "zap"`.

Case 4: system.defaults macOS tweaks

`system.defaults` is `nix-darwin`'s typed interface to `defaults write`. Every key maps to a known macOS preference domain. Values are type-checked at build time — a typo is a build error, not a silent no-op.

This module captures the settings the desk team applies to every macOS workstation. Save it as a standalone module imported by the main `flake.nix`.

Save as `macos-defaults.nix`:

```nix
# macos-defaults.nix
{ pkgs, lib, ... }:

{
  system.defaults = {

    #   Finder
    finder = {
      AppleShowAllExtensions = true;    # always show .go, .nix, .yaml suffixes
      AppleShowAllFiles      = true;    # show hidden dot-files
      ShowPathbar            = true;    # breadcrumb at bottom of every window
      ShowStatusBar          = true;    # file count and disk space in status bar
      FXPreferredViewStyle   = "Nlsv"; # default to list view
      FXDefaultSearchScope   = "SCcf"; # search current folder by default
      _FXSortFoldersFirst    = true;   # directories appear before files
    };

    #   Keyboard
    NSGlobalDomain = {
      ApplePressAndHoldEnabled = false; # disable character picker; enables key repeat
      InitialKeyRepeat         = 15;   # 225 ms before repeat starts (default: 25)
      KeyRepeat                = 2;    # 30 ms between repeats      (default: 6)
    };

    #   Dock
    dock = {
      autohide                = true;
      autohide-delay          = 0.0;
      autohide-time-modifier  = 0.2;
      show-recents            = false;   # no "recent apps" section
```

```
      tilesize                = 48;
      minimize-to-application = true;    # minimise into app icon, not Dock shelf
      mru-spaces              = false;   # keep Space order fixed
    };

    #   Screenshots
    screencapture = {
      location      = "/Users/alice/Screenshots";
      type          = "png";
      disable-shadow = true;
    };

    #   Trackpad
    trackpad = {
      Clicking              = true;   # tap-to-click
      TrackpadThreeFingerDrag = true;
    };
  };

  # Suppress .DS_Store on network and USB volumes via activation script
  system.activationScripts.extraActivation.text = ''
    defaults write com.apple.desktopservices DSDontWriteNetworkStores -bool true
    defaults write com.apple.desktopservices DSDontWriteUSBStores     -bool true
  '';
}
```

Import it from your flake:

```
# flake.nix  (modules list excerpt)
modules = [
  ./macos-defaults.nix
  ({ pkgs, ... }: {
    environment.systemPackages = [ pkgs.ripgrep pkgs.jq pkgs.git ];
    system.stateVersion = 6;
    system.primaryUser = "deskadmin";
  })
];
```

Apply:

```
sudo darwin-rebuild switch --flake .#desk-mac
```

Output:

```
building the system configuration...
system defaults applied:
  finder.AppleShowAllExtensions = true
  finder.ShowPathbar = true
```

```
  dock.autohide = true
  NSGlobalDomain.KeyRepeat = 2
  NSGlobalDomain.InitialKeyRepeat = 15
reloading Dock...
reloading Finder...
/run/current-system -> /nix/store/2njw8x9q...-darwin-system
```

Preferences take effect immediately. `InitialKeyRepeat` and `KeyRepeat` take effect after the next login.

The trap

`nix-darwin` does not replace the macOS kernel. It manages user space. Developers who have used NixOS sometimes reach for `boot.kernelModules`, `boot.kernelParams`, or `hardware.*` options in a `darwinConfiguration`. These options do not exist on Darwin.

Attempting it:

```nix
# flake.nix  ← WRONG: kernel options are Linux-only
darwinConfigurations."desk-mac" = nix-darwin.lib.darwinSystem {
  system = "aarch64-darwin";
  modules = [
    ({ ... }: {
      boot.kernelModules = [ "wireguard" ];  # does not exist on Darwin
    })
  ];
};
```

Run:

```
sudo darwin-rebuild switch --flake .#desk-mac
```

Output:

```
error: The option `boot.kernelModules' does not exist. Definition values:
- In anonymous module at `/nix/store/.../flake.nix':
    [ "wireguard" ]
(use '--show-trace' to show detailed location information)
```

The error fires at **build time**, before any change touches the live system.

The fix: On macOS, kernel extensions must be installed through Apple-signed packages or system recovery. WireGuard on macOS runs as a userspace process (`wireguard-go`), not a kernel module. Declare the `wireguard-go` binary in `environment.systemPackages` instead.

Similarly, SIP (System Integrity Protection) prevents `nix-darwin` from writing to `/System`, `/usr` (except `/usr/local`), or `/sbin`. Do not attempt to place files in those paths. Everything belongs in `/nix/store` and `/run/current-system`, which SIP does not protect.

The boring rule

- One `flake.nix` holds both `nixosConfigurations` and `darwinConfigurations`.
- Share tooling via a `commonPackages` function; append OS-specific packages in each configuration.
- Use `system.defaults` for every preference that would otherwise be a `defaults write` command.
- Run Home Manager as a `nix-darwin` module (`home-manager.darwinModules.home-manager`) for per-user dot-file management.
- Use `nix-homebrew` to manage Homebrew casks declaratively; set `onActivation.cleanup = "zap"` so removed casks are uninstalled automatically.
- Run `darwin-rebuild switch --flake .#<hostname>` after every change. No change is complete until the switch succeeds.
- Pin `nix-darwin/nix-darwin-26.05`, not `LnL7/…/master`. Set `system.primaryUser`. `stateVersion` is an integer (birth).
- Do not fight SIP. If an operation requires disabling SIP, it belongs in a one-time provisioning script during machine enrollment — not in `nix-darwin`.
- `sudo darwin-rebuild`. Do not set `nix.package = pkgs.nix` if that would downgrade a 2.35+ daemon to channel 2.34.

Try this

1. **Add a cask.** Add `"rectangle"` (a window manager) to `homebrew.casks` in Case 3's flake and run `darwin-rebuild switch`. Confirm `Rectangle.app` appears in `/Applications`. Then remove it from the list, switch again, and verify it is gone.

2. **Shared package drift.** Fork Case 2's flake. Add `pkgs.kubectl` to `commonPackages`. Apply to both `desk-server` (via `nixos-rebuild switch`) and `desk-mac` (via `darwin-rebuild switch`). Confirm `kubectl version --client` returns the same version string on both machines.

3. **Defaults archaeology.** On a macOS machine, run `defaults read com.apple.dock | grep autohide` and note the current value. Add `system.defaults.dock.autohide = true` to Case 1's flake, switch, and run the same `defaults read` command again to confirm the value changed.

4. **Trigger the trap deliberately.** In a `darwinConfiguration`, add `boot.kernelModules = [ "wireguard" ]` and run `darwin-rebuild switch`. Read the error message. Then replace it with `environment.systemPackages = [ pkgs.wireguard-go ]`, switch again, and confirm `wg-quick` is now on `$PATH`.

Nix on a Foreign Linux

Nix on a Foreign Linux

Most of the desk will not reinstall the OS this quarter. The boring default is: **install the Nix daemon on the existing Ubuntu/Fedora/Arch machine, use flakes + Home Manager + direnv, and leave `apt`/`dnf` for the kernel, firmware, and the desktop the vendor already glued together**.

This is not NixOS. `/etc`, systemd units, and the kernel stay the host's. You steal reproducible **user** environments and **project** shells. When you are ready to own the machine, install NixOS.

Mental model

Layer	Owner on a foreign distro
Kernel, glibc of `/usr`, display manager	Distro (`dnf`, `apt`)
`/nix/store`, `nix-daemon`	Nix 2.35+
User CLIs, git, prompt, nvim	Home Manager (standalone), `release-26.05`
Per-repo compilers	`nix develop` / direnv

`PATH` will contain **both** `/usr/bin` and `~/.nix-profile/bin`. Put the Nix profile **first** so `jq` is the one you pinned.

Do not replace `/usr/bin/python3` with a Nix python. The distro's tools (PackageKit, GNOME extensions, vendor installers) expect the distro python.

```
/usr/bin/…          distro, leave it
~/.nix-profile/bin  Home Manager
$PRJ/.direnv        project shell (direnv)
```

There is no `nixos-rebuild` here. `configuration.nix` does nothing until the machine is NixOS.

Worked examples

Case 1: Daemon is already installed — use it

After the Installing Nix chapter:

```
systemctl is-active nix-daemon
nix --version
```

```
echo "$PATH" | tr ':' '\n' | head
```

You want `nix (Nix) 2.35…` (or newer). If `~/.nix-profile/bin` is missing from `PATH`, your login shell did not source the daemon profile. For bash, until Home Manager owns it:

```
# ~/.bashrc
. /nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh
```

Confirm flakes in the **daemon** `nix.conf` (`experimental-features = nix-command flakes`), not only in `~/.config/nix/nix.conf` — multi-user builds ignore the user file.

Case 2: Standalone Home Manager on Fedora/Ubuntu

Save as `flake.nix` (user flake):

```
# flake.nix
{
  description = "Deskadmin on a foreign Linux";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  inputs.home-manager.url = "github:nix-community/home-manager/release-26.05";
  inputs.home-manager.inputs.nixpkgs.follows = "nixpkgs";

  outputs = { self, nixpkgs, home-manager }: {
    homeConfigurations.deskadmin = home-manager.lib.homeManagerConfiguration {
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
      modules = [
        {
          home.username = "deskadmin";
          home.homeDirectory = "/home/deskadmin";
          home.stateVersion = "26.05";
          home.packages = [ nixpkgs.legacyPackages.x86_64-linux.jq ];
        }
      ];
    };
  };
}
```

```
nix run home-manager/release-26.05 -- switch --flake .#deskadmin
which jq
```

`home.username` / `home.homeDirectory` must match `id -un` and `$HOME`. Home Manager will not create `/home/deskadmin` if you invented a name.

Case 3: Project shells without touching `/usr`

Save as `flake.nix` in a Go service repo:

```
# flake.nix
{
  description = "Desk metrics on a foreign Linux";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [ pkgs.go pkgs.gopls ];
        env.GOTOOLCHAIN = "local";
      };
    };
}
```

```
nix develop --command which go
which go || echo 'host go missing (good)'
```

Inside: store path. Outside: `/usr/bin/go` or nothing. That split is expected. direnv (`use flake`) makes the project one automatic.

Case 4: SELinux and `/nix`

On Fedora, SELinux can deny the daemon. If `nix-build` fails with `Permission denied` and `dmesg` mentions `nix-daemon`:

```
sudo ausearch -m avc -ts recent | head
```

Follow the distro package docs when you installed Nix via `dnf`. If you used the official installer, a systemd unit plus the installer-provided policy is required. Do not `setenforce 0` as a lifestyle.

Case 5: Uninstall without wrecking the distro

```
sudo systemctl stop nix-daemon
# official installer ships an uninstall script; use it
sudo rm -rf /nix
```

Then remove the `PATH` snippet from `~/.bashrc`. Distro packages (`vim`, `firefox`) stay. That is the whole point of not replacing `/usr`.

The trap

The trap is **`environment.systemPackages` thinking** on Ubuntu: writing a `configuration.nix` and wondering why `nixos-rebuild` is not installed. There is no NixOS module system on this host. Home Manager + flakes. If you want `services.openssh`, install NixOS.

The other trap is mixing distro Node and Nix Node on the same `PATH` without direnv. CI uses Nix; your laptop uses `/usr/bin/node`; lockfiles disagree.

The boring rule

- Nix daemon 2.35+ + flakes + Home Manager `release-26.05` + direnv. Distro owns the OS.
- Nix `PATH` first for **your** CLIs. Distro python/node stay for **vendor** tools.
- No `configuration.nix` until the machine is NixOS.
- SELinux: use the distro's Nix policy, not `setenforce 0`.
- Project compilers only from `nix develop`.

Try this

1. `which jq` before and after Home Manager with `jq` in `home.packages`. The second should be `/nix/store/…`.
2. Case 3: `nix develop --command which go` versus `which go` in a fresh terminal.
3. `echo $PATH` and label each entry as distro / Nix profile / direnv / other.
4. Confirm `systemctl is-enabled nix-daemon` is `enabled` so a reboot still builds.

Vendor Binaries with nix-ld and FHS Envs

Vendor Binaries with nix-ld and FHS Envs

Nix binaries look for `ld-linux` and `libssl.so` under `/nix/store`. A vendor AppImage, a JetBrains extractor, or a `curl | tar` toolchain looks for `/lib64/ld-linux-x86-64.so.2`. The boring default is: **`programs.nix-ld` for random ELF files, `buildFHSEnv` for programs that require an FHS tree, and never a global `LD_LIBRARY_PATH`**.

Mental model

Three kinds of program live on a desk:

Kind	Loader	Fix
Nix package	Store `ld-linux` + `RUNPATH`	None. Already correct.
Vendor ELF	Hard-coded `/lib64/ld-linux…`	`nix-ld` intercepts that path
"Must see `/usr/lib`" installer	FHS layout	`buildFHSEnv` / `steam-run`

`nix-ld` installs a **compatibility loader** at the path vendor binaries request. That loader then finds libraries from a Nix-provided list (`programs.nix-ld.libraries`). The vendor file stays where you downloaded it; it does not enter `/nix/store` unless you package it.

`buildFHSEnv` builds a **bubblewrapped** `/usr` with the packages you listed. The process sees `libc.so` in the usual places. Use it when `nix-ld` is not enough (scripts that `dlopen("/usr/lib/libfoo.so")`).

```
vendor binary
   requested interpreter: /lib64/ld-linux-x86-64.so.2


   nix-ld stub at that path


   real libraries from programs.nix-ld.libraries  (store paths)
```

Worked examples

Case 1: Enable nix-ld on NixOS

Save as `nix-ld.nix`:

```
# nix-ld.nix
{ pkgs, ... }:

{
  programs.nix-ld.enable = true;
  programs.nix-ld.libraries = with pkgs; [
    stdenv.cc.cc
    zlib
    openssl
    curl
    icu
    nss
    nspr
    libxml2
    libGL
    xorg.libX11
  ];
}
```

```
sudo nixos-rebuild switch
ls -l /lib64/ld-linux-x86-64.so.2
```

Output (shape):

```
lrwxrwxrwx 1 root root … /lib64/ld-linux-x86-64.so.2 -> /nix/store/…-nix-ld-…/lib/ld-linux-x86-
```

A vendor `./desk-toolbox` that died with `No such file or directory` (missing interpreter) should now start — or fail on a **named** missing library, which you add to `libraries`.

On **aarch64** the stub is `/lib/ld-linux-aarch64.so.1` (not `/lib64/ld-linux-x86-64.so.2`). `file ./desk-toolbox` must match the host. An x86_64 AppImage on ARM is qemu, not nix-ld.

```
file ./desk-toolbox
readelf -l ./desk-toolbox | grep interpreter
```

Case 2: See what is still missing

```
ldd ./desk-toolbox | grep 'not found'
```

Output (shape):

```
    libfoo.so.1 => not found
```

Find it in nixpkgs, add `pkgs.foo` to `programs.nix-ld.libraries`, switch, retry. Do not `export LD_LIBRARY_PATH=/nix/store/…/lib`.

`NIX_LD` / `NIX_LD_LIBRARY_PATH` are what the stub loader consults. `nix-ld` sets them from `programs.nix-ld.libraries` via `environment.sessionVariables`. A user export of `NIX_LD_LIBRARY_PATH` in `.bashrc` **overrides** the module list and is the same class of bug as `LD_LIBRARY_PATH`. Keep it in the module.

```
echo "$NIX_LD"
echo "$NIX_LD_LIBRARY_PATH" | tr ':' '\n' | head
```

Those paths must be store paths from `programs.nix-ld.libraries`, not `~/.local/lib`.

Case 3: `buildFHSEnv` for an installer that shells out

Save as `fhs.nix`:

```
# fhs.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.buildFHSEnv {
  name = "desk-fhs";
  targetPkgs = pkgs: with pkgs; [
    stdenv.cc.cc
    zlib
    openssl
    curl
    bash
    coreutils
    gnused
    gawk
  ];
  runScript = "bash";
}
```

```
nix-build fhs.nix
./result/bin/desk-fhs
```

You are now in a shell where `/usr/bin/env` and `libc.so.6` exist in FHS locations. Run the vendor installer **inside** this shell. Exit, and the host is unchanged.

`steam-run` is a pre-built FHS env in nixpkgs for the same trick:

```
nix run nixpkgs#steam-run -- ./desk-toolbox
```

Case 4: Package the vendor blob only when you must

If the team will run this for years, write a derivation with `autoPatchelfHook` instead of asking every laptop to enable nix-ld for it. That is the packaging part. nix-ld is the **escape hatch** for binaries you will not package this week.

```
# vendor-patchelf.nix fragment
{
  nativeBuildInputs = [ autoPatchelfHook ];
  buildInputs = [ stdenv.cc.cc zlib openssl ];
}
```

`autoPatchelfHook` rewrites the interpreter and RUNPATH **in the store copy**. The original download on disk is unchanged. That is the opposite of nix-ld (loader stays `/lib64`, libraries from the module). Pick one per binary.

Case 5: Foreign Linux

`programs.nix-ld` is a **NixOS** module. On Ubuntu, the equivalent is the `nix-ld` package plus a loader in `/lib64`, which fights the distro. Prefer `buildFHSEnv` / `steam-run` / `nix develop` on foreign distros, or run the vendor app in a container. Do not overwrite Ubuntu's `ld-linux`.

The trap

The trap is `export LD_LIBRARY_PATH=$HOME/libs` so one AppImage starts. Every Nix binary in that terminal now prefers `$HOME/libs/libssl.so`. The next day, `curl` speaks a surprise TLS stack.

The other trap is adding `pkgs.linuxPackages.kernel` to `nix-ld.libraries` "to have everything." Keep the list short. Add libraries when `ldd` names them.

A third: `NIX_LD_LIBRARY_PATH` in Home Manager `home.sessionVariables` fighting `programs.nix-ld`. A fourth: nix-ld **and** `autoPatchelfHook` on the same blob so RUNPATH and the stub disagree.

The boring rule

- Nix packages stay Nix packages. Do not wrap them in FHS.
- Vendor ELFs on NixOS: `programs.nix-ld` + a **short** library list from `ldd`. Match arch.
- Vendor installers that need `/usr`: `buildFHSEnv` or `steam-run`, not a fake `/usr` on the host.
- Never a global `LD_LIBRARY_PATH` or a profile `NIX_LD_LIBRARY_PATH`.
- `autoPatchelfHook` when the blob is worth a derivation; nix-ld is the week-one hatch. One or the other.
- If you run it daily, package it with `autoPatchelfHook` later.

Try this

1. On a NixOS VM, enable Case 1, download a random upstream `rg` static/dynamic release (not from nixpkgs), `file` it, and run it. If it is static, nix-ld is unused (fine). If dynamic, `ldd` it.
2. `ldd $(readlink -f $(which curl))` on a **Nix** curl — every line should already be a store path. nix-ld must not be involved.
3. Enter the FHS env from Case 3, run `ls /usr/bin | head` and `ldd /usr/bin/bash`. Exit and confirm `/usr/bin/bash` on the host is the NixOS one (or absent), not the bubble.
4. Add one library to `nix-ld.libraries`, switch, and `nix why-depends /run/current-system pkgs.openssl` to see that the system closure grew. That is the cost of a long list.

devenv: Modules, Services, and Processes

devenv: Modules, Services, and Processes

`pkgs.mkShell` plus `flake.lock` plus direnv is still the desk default for a compiler and an LSP. **devenv** is the next layer: a NixOS-style *module system for the developer environment* — languages, processes, and local databases — with its own CLI and lock. Use it when `mkShell { packages = [ pkgs.go pkgs.gopls ]; }` is no longer enough. Do not use it to replace `buildGoModule` or to pin `nixpkgs` to a rolling branch.

This chapter is devenv **2.x** (native process manager, optional `git-hooks` input). Commands below assume a Nix 2.35+ daemon.

Mental model

devenv evaluates `devenv.nix` (modules) against inputs in `devenv.yaml`, pins them in `devenv.lock`, and exposes a shell (`devenv shell`) plus a process supervisor (`devenv up`).

```
devenv.yaml   →  inputs (nixpkgs, extra flakes)
devenv.nix    →  packages, languages.*, services.*, processes.*, env
devenv.lock   →  rev + narHash  (commit this)

          devenv shell     PATH, env, enterShell
          devenv up        postgres, redis, desk-api, …


          $DEVENV_STATE     service data (not in git)
```

Need	Tool
`go` + `gopls` only	`mkShell` + `flake.lock` (earlier chapters)
Language module (LSP, formatter, version knob)	`languages.go.enable`
Local Postgres / Redis like Compose	`services.postgres` / `services.redis` + `devenv up`
Ad-hoc long-running command	`processes.<name>.exec`
Same env as a flake `devShell`	`devenv.lib.mkShell` — **impure** (`--no-pure-eval`)

`$DEVENV_ROOT` is the project directory. `$DEVENV_STATE` is where Postgres writes its cluster. Changing `initialScript` does **not** migrate an existing cluster — delete that state dir on purpose.

devenv 2.x drives processes with a **native** Rust manager. `process.manager.implementation = "process-compose";` is a transition flag, not the default.

Worked examples

Case 1: Init, pin 26.05, Go toolchain

```
nix run github:cachix/devenv -- init
```

- Creating devenv.nix
- Creating devenv.yaml
- Creating .gitignore

Overwrite the generated `devenv.yaml` so nixpkgs is **this book's train**, not `cachix/devenv-nixpkgs/rolling`:

```
# devenv.yaml
inputs:
  nixpkgs:
    url: github:NixOS/nixpkgs/nixos-26.05
```

Save as `devenv.nix`:

```
# devenv.nix
{ pkgs, ... }:

{
  packages = [ pkgs.git pkgs.jq ];

  languages.go.enable = true;
  # 26.05's pkgs.go; do not also set a random languages.go.package unless you measured it

  env.GOTOOLCHAIN = "local";
  env.CGO_ENABLED = "0";
  env.GOPRIVATE = "git.desk.internal";

  enterShell = ''
    echo "desk-api devenv: $(go env GOVERSION) GOTOOLCHAIN=$(go env GOTOOLCHAIN)"
  '';
}
```

```
devenv update
git add devenv.nix devenv.yaml devenv.lock .gitignore
devenv shell -- go version
```

Output (shape — versions follow 26.05):

```
desk-api devenv: go1.24.x GOTOOLCHAIN=local
go version go1.24.x linux/amd64
```

`devenv.lock` is the pin. `devenv update` without a lock commit is a channel. `languages.go.enable` pulls the language module (compiler, typically `gopls` / tools — inspect with `devenv info`). Extra CLIs still go in `packages`.

Do **not** put `pkgs.go` on Home Manager *and* `languages.go` here. One compiler, this project.

Case 2: Postgres + API process (`devenv up`)

This is why devenv exists on the desk: a local Postgres that is not Docker Desktop, plus the API in the same supervisor.

```
# devenv.nix
{ pkgs, lib, config, ... }:

{
  languages.go.enable = true;
  env.GOTOOLCHAIN = "local";

  services.postgres = {
    enable = true;
    package = pkgs.postgresql_16;
    listen_addresses = "127.0.0.1";
    port = 5432;
    initialDatabases = [{ name = "desk"; }];
    initialScript = ''
      CREATE USER desk WITH PASSWORD 'desk-dev';
      GRANT ALL PRIVILEGES ON DATABASE desk TO desk;
    '';
  };

  env.DATABASE_URL = "postgres://desk:desk-dev@127.0.0.1:${toString config.services.postgres

  processes.desk-api = {
    exec = "go run ./cmd/desk-api";
    after = [ "devenv:processes:postgres" ];
  };
}
```

```
devenv up
```

```
postgres   | database system is ready to accept connections
desk-api   | listening on :8080
```

Stop with Ctrl-C or `devenv processes down` / `devenv down` (2.x). Background: `devenv up -d`.

Dev password stays in devenv. Production is sops `EnvironmentFile`. After you change `initialScript`, wipe state or Postgres will ignore it:

```
rm -rf "$DEVENV_STATE/postgres"
devenv up
```

`postgresql_16` is whatever 26.05 named; if the attr is missing, `nix search github:NixOS/nixpkgs/nixos-26.05 postgresql`. Do not pin `postgresql_15` from a blog.

Readiness: `after = [ "devenv:processes:postgres" ]` waits until the **process** is considered ready, not until `pg_isready` unless the service module defines a probe. Check with `devenv processes` / logs. For HTTP:

```
{
  processes.desk-api = {
    exec = "go run ./cmd/desk-api";
    after = [ "devenv:processes:postgres" ];
    process-compose.readiness_probe.http_get = {
      host = "127.0.0.1";
      port = 8080;
      path = "/healthz";
    };
  };
}
```

On devenv 2.x the native manager has its own probe fields (`process-compose.*` is the old name if you set `process.manager.implementation = "process-compose"`). Prefer the native manager; read `devenv info` / the option reference for your lock if a field eval-errors.

Case 3: Extra inputs, git-hooks, lock hygiene

devenv 2.x does **not** include `git-hooks` by default. If you want hooks, add the input.

```
# devenv.yaml
inputs:
  nixpkgs:
    url: github:NixOS/nixpkgs/nixos-26.05
  git-hooks:
    url: github:cachix/git-hooks.nix
```

```
# devenv.nix fragment
{
  git-hooks.hooks = {
    nixfmt-rfc-style.enable = true;
  };
}
```

```
devenv update git-hooks
devenv shell   # installs hooks into .git/hooks via a task
```

```
git commit --allow-empty -m 'hook smoke'
```

`pre-commit` was renamed toward `prek` in 2.x. If eval says the hook attr moved, follow the error — do not copy a 1.x gist.

One-input updates:

```
devenv update nixpkgs
```

That is `nix flake update nixpkgs` for this tree. Do not `devenv update` of everything on Friday.

`cachix.pull` defaults to the devenv cache. The desk still uses **one team cache** for `desk-api` NARs. Pulling `devenv.cachix.org` is optional sugar for devenv's own closures; it is not a substitute for signing what you ship.

Case 4: direnv — `use devenv`, not a second `mkShell`

devenv's generated `.envrc` (or):

```
# .envrc
eval "$(devenv direnvrc)"
use devenv
watch_file devenv.nix
watch_file devenv.yaml
watch_file devenv.lock
```

```
direnv allow
go env GOVERSION
```

On `cd`, direnv loads the devenv shell. `devenv up` then uses that cache (faster; skips a full re-eval). Gitignore `.devenv/` and `$DEVENV_STATE` (devenv's `.gitignore` already should). Commit `.envrc`.

Do **not** also `use flake` in the same directory pointing at a competing `devShells.default`. One loader.

Named profiles (2.x) if backend vs frontend:

```
# devenv.nix fragment
{
  profiles.backend.module = {
    services.postgres.enable = true;
    languages.go.enable = true;
  };
  profiles.frontend.module = {
    languages.javascript.enable = true;
  };
}
```

```
devenv --profile backend shell
devenv --profile backend up
```

That is devenv's answer to `devShells.backend`. Prefer it over two ad-hoc flakes if you already paid for devenv.

Case 5: Flake wrapper (when CI already speaks `nix develop`)

Keep `devenv.nix` as source of truth. Export a `devShell` for tooling that only knows flakes:

```
# flake.nix
{
  description = "desk-api via devenv (optional flake wrap)";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    devenv.url = "github:cachix/devenv";
    devenv.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, devenv, ... }@inputs:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      devShells.${system}.default = devenv.lib.mkShell {
        inherit inputs pkgs;
        modules = [ ./devenv.nix ];
      };
    };
}
```

```
nix develop --no-pure-eval
```

`--no-pure-eval` is required. Flake pure eval cannot see the project directory (`devenv.root`). Forgetting the flag is a confusing eval error, not a missing package. An alternative is hard-coding `devenv.root = /absolute/path;` — that flake is not portable. Prefer the flag, or skip the wrapper and call `devenv shell` in CI:

```
# .github/workflows/check.yml fragment
- run: nix run github:cachix/devenv -- test
```

`devenv test` / `enterTest` runs tasks. Starting processes during flake-based `nix flake check` is **not** supported the same way — `devenv up` is the CLI. Do not pretend `flake check` boots Postgres.

`languages.*.import` / `devenv build` **outputs** exist (crate2nix, uv2nix, …). The desk still ships **`buildGoModule` + `vendorHash`** for `desk-api`. devenv outputs are an extra packaging stack; they are not the boring production path.

The trap

The trap is **`inputs.nixpkgs.url: github:cachix/devenv-nixpkgs/rolling`** (the init default) while production flakes pin `nixos-26.05`. Laptop `go` and CI `go` diverge. Pin `github:NixOS/nixpkgs/nixos-26.05` in `devenv.yaml` and commit `devenv.lock`.

The other traps:

- **devenv as the package builder.** `devenv shell` is PATH. `nix build .#desk-api` is the artifact.
- **`nix develop` without `--no-pure-eval`** on a `devenv.lib.mkShell` flake.
- **Two supervisors:** `docker compose up` *and* `devenv up` for the same Postgres. Pick one. Compose stays for the weird native addon; devenv `services.postgres` is the Nix-shaped default.
- **Leaving `$DEVENV_STATE/postgres` forever** after a major/package bump. Cluster files from 15 will not start 16.
- **`git-hooks` copied from a 1.x gist** without the `git-hooks` input on 2.x.
- **`languages.go.enable` plus `home.packages = [ pkgs.go ].`** Two toolchains.

The boring rule

- `mkShell` + `flake.lock` + direnv for a compiler. devenv when you need **languages / services / processes**.
- `devenv.yaml` nixpkgs = **`nixos-26.05`**. Commit `devenv.lock`. Update one input per PR.
- `devenv shell` / direnv `use devenv`. `devenv up` for Postgres and the API. State in `$DEVENV_STATE`, not git.
- Native process manager (2.x default). process-compose only if a probe option you need is still there.
- Production packages: `buildGoModule` (etc.), not `devenv build` outputs.
- Flake wrap is optional and **impure** (`--no-pure-eval`). CI can call `devenv test` instead.
- One Go, one Postgres, one loader. No Compose + devenv + mkShell fighting.

Try this

1. Case 1: `devenv init`, rewrite `devenv.yaml` to 26.05, `devenv update`, `devenv shell -- go env GOVERSION GOTOOLCHAIN`. Confirm a store path `which go`.
2. Case 2: `devenv up`, `psql "$DATABASE_URL" -c 'SELECT 1'`. Change `initialScript`, restart without wiping state, observe it did nothing; `rm -rf "$DEVENV_STATE/postgres"` and retry.
3. `git check-ignore -v devenv.lock` — must **not** be ignored. `git grep rolling -- devenv.yaml` — empty.

4. Add `use devenv` to `.envrc`, `direnv allow`, `cd` out and back. Confirm `go` stays a store path.
5. Optional: wrap Case 5, run `nix develop` **without** `--no-pure-eval`, read the error, then with the flag.

Packages and Overlays

The Nix Package Manager in Practice

The Nix Package Manager in Practice

Nix the language is useless without Nix the tool. The boring default on Nix 2.35 is: **`nix shell` for five minutes, `nix profile` only for personal CLIs that are not a project, and `nix search` against a 26.05 pin — not `nix-env -iA`.**

Project compilers still belong in `devShell` / direnv. This chapter is the **user** profile and one-shot PATH.

Mental model

`~/.nix-profile` is a generation of symlinks into `/nix/store`. It is not `/usr/bin`. Adding `jq` does not delete the previous `jq`; rollback is another generation.

Command	Lifetime
`nix shell nixpkgs#jq`	This process
`nix run nixpkgs#jq -- --version`	One command
`nix profile install nixpkgs#jq`	Until rollback/remove
`nix search nixpkgs jq`	Query

Unqualified `nixpkgs#` follows your **registry** (often a moving flake). For desk work prefer a pinned flake:

```
nix shell github:NixOS/nixpkgs/nixos-26.05#jq
```

or `nix shell .#jq` inside a repo.

Worked examples

Case 1: One-shot

```
nix shell github:NixOS/nixpkgs/nixos-26.05#cowsay --command cowsay "Desk system ready"
```

Output:

```
 --------------------
< Desk system ready >
 --------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Nothing new in `nix profile list`.

Case 2: `nix run`

```
nix run github:NixOS/nixpkgs/nixos-26.05#hello
```

Output:

```
Hello, world!
```

`run` builds (or substitutes) and executes the default app. Good for `nix run .#desk-api`.

Case 3: Profile install + list

```
nix profile install github:NixOS/nixpkgs/nixos-26.05#jq
nix profile list
which jq
readlink -f "$(which jq)"
```

Output (shape):

```
Name:               jq
Flake attribute:    packages.x86_64-linux.jq
Store path:         /nix/store/…-jq-…
```

`readlink -f $(which jq)` is a store path. On NixOS, prefer Home Manager `home.packages` over a growing profile if the user already has HM.

Case 4: Rollback

```
nix profile history
nix profile rollback
which jq || echo 'jq gone'
```

If `jq` was the only package in that generation, it disappears from PATH. The store path remains until GC.

```
nix profile remove jq   # if your 2.35 CLI uses names; else remove by index from `nix profil
```

Case 5: Search against 26.05, not the registry drift

```
nix search github:NixOS/nixpkgs/nixos-26.05 jq
```

Output (shape):

```
* legacyPackages.x86_64-linux.jq
  jq: Command-line JSON processor
```

First search can be slow (eval). After that, 26.05 is cached. `nix search nixpkgs jq` without a pin is whatever the flake registry says today.

Pin the user registry so unqualified `nixpkgs#` matches the desk:

```
nix registry pin nixpkgs github:NixOS/nixpkgs/nixos-26.05
nix registry list
```

CI still uses the **repo** `flake.lock`, not your laptop registry. The pin only stops surprise `nix shell nixpkgs#jq` on a workstation.

```
nix profile upgrade --all
```

On Nix 2.35 this rebuilds profile packages against whatever they were installed from. Prefer deleting a profile package and using Home Manager / the project flake instead of living on `upgrade --all`. Compilers still do not belong here.

The trap

The trap is **`nix-env -iA nixpkgs.jq`**. Unpinned, fights flakes, surprising rollbacks. `nix-env` is legacy. `nix profile` or Home Manager.

The other trap is `nix profile install` of `go`, `nodejs`, `rustc`. The next repo wants another version. Put those in the project `devShell`.

A third trap: `nix shell nixpkgs#jq` with an **unlocked** registry while CI uses 26.05. Pin the URL or the repo flake.

The boring rule

- `nix shell` / `nix run` for trials. Pin `nixos-26.05`.
- `nix profile` for personal CLIs (`jq`, `ripgrep`). Not for compilers.
- `nix profile rollback` when an install was a mistake.
- No `nix-env`.
- Project tools in the flake, not in the user profile.

Try this

1. `nix shell github:NixOS/nixpkgs/nixos-26.05#fastfetch --command fastfetch` (or `hello`).
2. `nix search github:NixOS/nixpkgs/nixos-26.05 postgresql | head`.
3. Install `hello` into the profile, run it, rollback, confirm `which hello`.
4. `nix profile list` and remove anything that is a compiler.

Understanding the Flakes Input System

Understanding the Flakes Input System

Two humans, two nixpkgs, two glibcs. The boring default is: **`inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05"`**, **`follows` so nested flakes share it**, and **`flake.lock` committed**.

Mental model

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  home-manager = {
    url = "github:nix-community/home-manager/release-26.05";
    inputs.nixpkgs.follows = "nixpkgs";
  };
};
```

Piece	Job
`url`	Where to fetch (github, git+ssh, path, tarball)
`follows`	Reuse **this** flake's input instead of nested default
`flake.lock`	`rev` + `narHash` for every input

`nix flake update` moves **every** input. `nix flake update nixpkgs` moves one. Do not `update` everything on Friday without `flake check`.

`master` / `nixpkgs-unstable` is not a pin you can name in an incident. 26.05 is. Home Manager's train is `release-26.05`, not `master`.

Worked examples

Case 1: `follows` in the tree

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk infrastructure flake with shared nixpkgs";

  inputs = {
```

```
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    home-manager = {
      url = "github:nix-community/home-manager/release-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, home-manager }: {
    formatter.x86_64-linux = nixpkgs.legacyPackages.x86_64-linux.nixfmt-rfc-style;
  };
}
```

`flake-utils` is optional sugar; skip it unless you already depend on it.

```
nix flake lock
nix flake metadata
```

Output (shape):

```
Inputs:
  home-manager: github:nix-community/home-manager/…
      nixpkgs follows input 'nixpkgs'
  nixpkgs: github:NixOS/nixpkgs/nixos-26.05/…
```

`follows input 'nixpkgs'` is the line you want. Without it, `home-manager` may lock a second nixpkgs.

Case 2: One-input update

```
nix flake update home-manager
```

Output (shape):

```
• Updated input 'home-manager'
```

`nixpkgs` did not move. That is the point. A lock PR that also bumps sops, disko, and nixpkgs is three incidents in one.

Case 3: Audit the lock

```
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.locked.rev'
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.locked.narHash'
```

Put that rev in the PR description when you bump 26.05.

Case 4: `path:` for a sibling repo

```
# flake.nix fragment
{
  inputs.desk-tooling.url = "path:../desk-tooling";
  inputs.desk-tooling.inputs.nixpkgs.follows = "nixpkgs";
}
```

`path:` is not copied to a cache the same way as `git+ssh`. Fine on a laptop. CI should use a git URL + lock (`git+ssh://git.desk.internal/desk-tooling.git?ref=refs/tags/…`).

Case 5: Check after a bump

```
nix flake update nixpkgs
nix flake check -L
```

If check fails, `git checkout -- flake.lock` and try a smaller step. The lock is git; treat it as code.

```
git diff --stat flake.lock
```

If `flake.lock` is uncommitted, there is no team.

When you add sops / disko / home-manager, give each a `follows`:

```
# flake.nix fragment
{
  inputs.sops-nix.url = "github:Mic92/sops-nix";
  inputs.sops-nix.inputs.nixpkgs.follows = "nixpkgs";
  inputs.disko.url = "github:nix-community/disko";
  inputs.disko.inputs.nixpkgs.follows = "nixpkgs";
}
```

```
nix flake metadata | grep -c 'nixpkgs:'
```

One `nixpkgs:` line (plus `follows` children). Three unlocked nixpkgs nodes is the eval-RAM incident.

The trap

The trap is **three copies of nixpkgs** because nobody wrote `follows`. Eval RAM and download time explode. `nix flake metadata` every time you add an input.

The other trap is not committing `flake.lock`. CI then evals whatever `nixos-26.05` means this morning.

The boring rule

- `nixos-26.05` (and matching `release-26.05` for HM). Not `master`.
- `inputs.<dep>.inputs.nixpkgs.follows = "nixpkgs";`
- Commit the lock. Update one input per PR when you can.
- `metadata` to see follows. Count nixpkgs nodes: one.
- `path:` locally; git URLs in CI.

Try this

1. `nix flake metadata` on this flake; count nixpkgs nodes. Should be one.
2. Remove `follows`, `nix flake lock --update-input home-manager`, metadata again, put `follows` back.
3. `nix flake update home-manager` and `git diff flake.lock`.
4. `jq` the locked `narHash` for nixpkgs; paste it in a dummy PR description.

Creating and Composing Overlays

Creating and Composing Overlays

You need a patch, not a nixpkgs fork. The boring default is: **`final: prev: { … }`**, **`prev.pkg.override / overrideAttrs`**, **`final` for siblings**, and a **short** overlay — not a private nixpkgs.

Mental model

```
final: prev: {
  deskGreet = prev.writeShellScriptBin "desk-greet" "echo desk";
  hello = prev.hello.overrideAttrs (old: { pname = old.pname; });
}
```

Arg	Meaning
`prev`	Package set **before** this overlay. Override **this** package from `prev`.
`final`	Package set **after** all overlays. Use for *other* packages this overlay also defines.

`final.hello.overrideAttrs` when you meant `prev.hello` is infinite recursion.

Apply:

```
import nixpkgs { overlays = [ overlayA overlayB ]; }
```

or NixOS:

```
# configuration.nix fragment
{
  nixpkgs.overlays = [
    (import ./overlays/desk.nix)
  ];
}
```

List order is apply order. Later overlays see earlier ones through `final`. A third overlay that also sets `hello` wins if it uses `prev` from after the second — or recurses if it uses `final.hello`. Keep **one** overlay file per concern (`overlays/hello.nix`, `overlays/desk-scripts.nix`) and import the list.

Worked examples

Case 1: Add a script to pkgs

Save as `overlay_demo.nix`:

```
# overlay_demo.nix
let
  deskOverlay = final: prev: {
    deskGreet = prev.writeShellScriptBin "desk-greet" ''
      echo "Welcome to the desk infrastructure"
    '';
  };
  pkgs = import <nixpkgs> { overlays = [ deskOverlay ]; };
in
{
  name = pkgs.deskGreet.name;
}
```

```
nix eval --impure --file overlay_demo.nix --json
```

(`--impure` because `<nixpkgs>`. In a flake, pass `nixpkgs` from inputs — no impure.)

Output:

```
{"name":"desk-greet"}
```

Case 2: Compose two overlays

Save as `compose_overlays.nix`:

```
# compose_overlays.nix
let
  overlayA = final: prev: { servicePort = 8080; };
  overlayB = final: prev: {
    serviceUrl = "http://127.0.0.1:${toString final.servicePort}";
  };
  pkgs = import <nixpkgs> { overlays = [ overlayA overlayB ]; };
in
{ url = pkgs.serviceUrl; }
```

```
nix eval --impure --file compose_overlays.nix --apply 'x: x.url'
```

Output:

```
"http://127.0.0.1:8080"
```

`overlayB` reads `final.servicePort` so it sees overlay A's binding.

Case 3: `overrideAttrs` from `prev`

```
# hello-overlay.nix
final: prev: {
  hello = prev.hello.overrideAttrs (old: {
    pname = "desk-hello";
  });
}
```

The output path name changes. Downstream `pkgs.hello` is your patched one **if** they take `hello` from the same `pkgs`.

`pkg.override { enableFoo = true; }` is for **callPackage arguments**. `overrideAttrs` is for the derivation attrset (`postPatch`, `version`). Using `overrideAttrs` to flip a `callPackage` flag does nothing. Read the package file: if it is `{ stdenv, enableFoo ? false }:`, you want `override`.

Case 4: Infinite recursion (the foot-gun)

```
# bad-overlay.nix
final: prev: {
  hello = final.hello.overrideAttrs (old: { });
}
```

```
nix eval --impure --expr 'import <nixpkgs> { overlays = [ (import ./bad-overlay.nix) ]; }' -
```

You will get infinite recursion (or a hang/error). Use `prev.hello`.

Case 5: Flake overlay export

```
# flake.nix fragment
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }: {
    overlays.default = final: prev: {
      deskGreet = prev.writeShellScriptBin "desk-greet" "echo desk";
    };
  };
}
```

Consumers: `overlays = [ desk-tooling.overlays.default ];` with `follows` on nixpkgs.

The trap

The trap is `final.pkg.override…`. Recursion.

The other trap is a 2000-line overlay that copies half of nixpkgs. Pin a package with `fetchFromGitHub` + `callPackage` (next chapter) or an overlay of **one** attr. Fork nixpkgs only when you are actually contributing upstream.

The boring rule

- Signature `final: prev:`.
- Override **this** package from `prev`.
- Refer to **other** overlay packages via `final`.
- Keep overlays small. Files under `overlays/`.
- 26.05 `follows` so the overlay applies to one pkgs.

Try this

1. Overlay `desk-status` with `writeShellScriptBin`; `nix-build -E 'with import <nixpkgs> { overlays = [ … ]; }; desk-status'` (impure lab) or a flake package.
2. Two overlays: A sets `deskPort = 9;`, B builds a string from `final.deskPort`.
3. Write the bad `final.hello` overlay and trigger recursion; switch to `prev`.
4. `nix flake metadata` after adding an overlay-only flake input.

Custom Package Definitions

Custom Package Definitions

Internal CLIs need a recipe, not a gist in `$HOME`. The boring default is: **`{ lib, stdenv, … }:` + `pkgs.callPackage ./desk-tool.nix { }`, store-path wrappers**, and **no hardcoded `pkgs.curl` inside the recipe**.

Mental model

```
desk-tool.nix     function of dependencies
callPackage       fills args from pkgs
$out/bin/…        the artifact
```

`callPackage ./f.nix { curl = otherCurl; }` overrides one arg. That is how overlays and cross work.

Scripts: `writeShellApplication` (fundamentals) is enough for many desk tools. `stdenv.mkDerivation` when you compile. `makeWrapper` / `wrapProgram` when a script must see `curl` on PATH at **runtime**.

Worked examples

Case 1: `writeShellApplication` recipe

Save as `desk-tool.nix`:

```nix
# desk-tool.nix
{ writeShellApplication, curl, jq }:

writeShellApplication {
  name = "desk-tool";
  runtimeInputs = [ curl jq ];
  text = ''
    echo "Querying desk API with curl and jq..."
  '';
}
```

Save as `default.nix`:

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./desk-tool.nix { }
```

```
nix-build default.nix
./result/bin/desk-tool
```

Output:

```
Querying desk API with curl and jq...
```

`runtimeInputs` puts `curl` and `jq` on PATH inside the wrapper. The host need not have them.

Case 2: `stdenv` + `wrapProgram`

Save as `desk-tool-stdenv.nix`:

```
# desk-tool-stdenv.nix
{ lib, stdenv, makeWrapper, curl, jq }:

stdenv.mkDerivation {
  pname = "desk-tool";
  version = "1.0.0";
  src = ./.;
  dontUnpack = true;
  nativeBuildInputs = [ makeWrapper ];
  installPhase = ''
    mkdir -p $out/bin
    echo '#!/bin/sh' > $out/bin/desk-tool
    echo 'echo desk-tool' >> $out/bin/desk-tool
    chmod +x $out/bin/desk-tool
    wrapProgram $out/bin/desk-tool \
      --prefix PATH : ${lib.makeBinPath [ curl jq ]}
  '';
}
```

Prefer Case 1 unless you already have an `installPhase`.

Case 3: Override an input

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./desk-tool.nix {
```

```
  jq = pkgs.jq;
}
```

Passing `jq` explicitly is how you inject a patched jq from an overlay.

Case 4: Flake package

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      packages.x86_64-linux.desk-tool = pkgs.callPackage ./desk-tool.nix { };
      packages.x86_64-linux.default = self.packages.x86_64-linux.desk-tool;
    };
}
```

```
nix build
./result/bin/desk-tool
```

Case 5: `meta`

```
meta = {
  description = "Internal desk operations utility";
  license = lib.licenses.mit;
  platforms = lib.platforms.unix;
};
```

`nix search` / `nix flake show` readers see this. Internal tools still get `meta`.

`passthru.tests` is how you attach a check without changing `$out`:

```
# desk-tool.nix fragment
passthru.tests.runs = pkgs.runCommand "desk-tool-runs" { } ''
  ${desk-tool}/bin/desk-tool >/dev/null
  touch $out
'';
```

Wire that into `checks` in the flake. `meta.mainProgram = "desk-tool";` makes `nix run .#desk-tool` work without guessing the bin name.

The trap

The trap is **`pkgs.curl` inside `desk-tool.nix`**. Then `callPackage` cannot swap curl. Cross-compile and overlays lose. Arguments on the function, always.

The other trap is `src = ./.;` including `.git` and `README` (caching chapter). `lib.fileset` / `cleanSource` when the recipe grows.

The boring rule

- Recipe is a function. `callPackage` injects.
- `writeShellApplication` for scripts; `stdenv` for builds.
- Runtime tools via `runtimeInputs` / `wrapProgram`, not the user's PATH.
- Flake `packages.<system>.desk-tool` on 26.05. `meta.mainProgram` for `nix run`.
- `passthru.tests` for checks; do not hide tests only in CI YAML.
- Override through `callPackage ./f.nix { curl = …; }`.

Try this

1. Add `hello` to `runtimeInputs` and call `hello` from the script.
2. `callPackage ./desk-tool.nix { jq = pkgs.jq; }` explicitly.
3. `nix path-info -Shr ./result` and find curl in the closure.
4. Break the shebang by using a raw `echo > $out/bin` without wrap; run on a PATH without curl; then restore the wrapper.

Managing Dependencies Across Projects

Managing Dependencies Across Projects

Five repos, five copies of `desk-checker`, five versions. The boring default is: **one tooling flake on nixpkgs 26.05 exporting `packages` and `overlays.default`; consumers `follows` that nixpkgs and pin the tooling input**.

This is the same idea as the platform-engineering IDP chapter, at **package** scale: a linter, not a whole OS.

Mental model

```
desk-tooling (flake)
    packages.*.desk-checker
    overlays.default


billing-service  inputs.desk-tooling
                 inputs.nixpkgs.follows = "desk-tooling/nixpkgs"
                 # or both follow a platform flake
```

`nix flake update desk-tooling` is the upgrade. Not curl | sh. The consumer's `flake.lock` records the tooling **rev**. CI must fetch that git URL, not `path:../tooling`.

Worked examples

Case 1: Tooling flake

Save as `tooling/flake.nix`:

```nix
# flake.nix
{
  description = "Shared desk tooling";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
```

```
      desk-checker = pkgs.writeShellApplication {
        name = "desk-checker";
        text = ''
          echo "Validating desk organizational standards: OK"
        '';
      };
    in
    {
      packages.${system}.desk-checker = desk-checker;
      packages.${system}.default = desk-checker;
      overlays.default = final: prev: {
        desk-checker = self.packages.${prev.stdenv.hostPlatform.system}.desk-checker;
      };
    };
}
```

(If `hostPlatform.system` is awkward in an overlay, export only `packages` and reference `desk-tooling.packages.${system}.desk-checker` from the consumer — often clearer. That is Case 3, and the boring default.)

Case 2: Consumer with `follows`

Save as `project/flake.nix`:

```
# flake.nix
{
  description = "Billing service";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    desk-tooling.url = "path:../tooling";
    desk-tooling.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, desk-tooling }:
    let
      system = "x86_64-linux";
      pkgs = import nixpkgs {
        inherit system;
        overlays = [ desk-tooling.overlays.default ];
      };
    in
    {
      devShells.${system}.default = pkgs.mkShell {
        packages = [ pkgs.desk-checker ];
      };
```

```
    };
}
```

```
cd project
nix develop --command desk-checker
```

Output:

```
Validating desk organizational standards: OK
```

Case 3: Prefer packages over overlay

```
devShells.${system}.default = pkgs.mkShell {
  packages = [ desk-tooling.packages.${system}.desk-checker ];
};
```

No overlay, no `pkgs.desk-checker` magic. Overlay when many packages must look like nixpkgs (`callPackage` trees, `override`, `overrideAttrs`). A single checker does not earn an overlay. Overlays that replace `stdenv` or `python3` in a consumer shell are how you get two glibcs and a six-hour rebuild.

Case 4: Update the pin

```
nix flake update desk-tooling
nix flake check
git diff flake.lock
```

CI on the consumer fails if the tooling flake broke `desk-checker`. That is the contract. Review the lock diff: one input should move (`desk-tooling`), and nixpkgs should **not** move unless you meant `nix flake update` (everything). `follows` keeps them on the same nixpkgs; a lock that shows two nixpkgs revs means someone dropped `follows`.

A scheduled workflow that only runs `nix flake update desk-tooling && nix flake check` and opens a PR is the boring bump. Humans still merge it.

Case 5: CI uses git, not `path:`

```
# flake.nix fragment - CI and humans
{
  inputs.desk-tooling.url = "git+ssh://git@git.desk.internal/desk-tooling";
  # or: github:desk/desk-tooling
}
```

`path:../tooling` is for this chapter's laptop. CI checkouts one repo; `path:` to a sibling that is not in the workspace is a missing input. Lock the git rev in `flake.lock`. `nix flake metadata` must show a git rev, not a dirty path, on the runner.

Private git: runner `access-tokens` / SSH deploy key (CI secrets chapter), not a token in `flake.nix`.

The trap

The trap is **copying `desk-checker.sh` into every repo**. The fifth copy diverges. One flake.

The other trap is **not `follows`**. Tooling on 26.05, app on another nixpkgs, two glibcs in one shell.

A third is **overlays for one binary**. A fourth is `path:` in CI, or `nix flake update` (all inputs) when you meant `update desk-tooling`.

The boring rule

- One tooling flake. 26.05. Lockfile.
- Export `packages`; overlay only if many attrs must look like nixpkgs. Never overlay `stdenv` "for convenience."
- Consumers `follows` nixpkgs. One glibc in the shell.
- `nix flake update desk-tooling` is the bump. Review `flake.lock`.
- git URL in CI; `path:` only locally. Tokens stay in runner config.

Try this

1. Two directories as in Cases 1–2; `nix develop --command desk-checker`.
2. Change the echo text in tooling, `update` the consumer, run again.
3. `nix flake metadata` in the consumer; one nixpkgs.
4. Replace the overlay with a direct `packages` reference (Case 3).
5. `nix flake metadata` in CI logs: `desk-tooling` is a git rev. If it says `path:`, the workflow checked out the wrong tree.

Fetchers and Fixed-Output Derivations

Fetchers and Fixed-Output Derivations

Every package starts as bytes from somewhere. The boring default is: **`fetchurl` / `fetchFromGitHub` with an SRI `hash = "sha256-…"`, and a hash mismatch is a stop, not a prompt to disable the check**.

Mental model

A **fixed-output derivation** (FOD) may use the network. It must produce bytes whose hash you declared. The store path is a function of that hash. Mirrors can change; the hash must not.

Fetcher	When
`fetchurl`	A tarball or file at a URL
`fetchFromGitHub`	`owner` + `repo` + `rev` (tag or commit)
`fetchgit`	Arbitrary git remote
`fetchzip`	A zip you want unpacked
`fetchpatch` / `fetchpatch2`	One patch file (`fetchpatch2` on 26.05)

Language vendor hashes (`vendorHash`, `cargoHash`) are the same idea: FODs for Go modules / crates.io.

```
url + rev          hash you wrote


 download   cmp    sha256   ok    /nix/store/<fod-hash>-src

                      fail → error: hash mismatch (got …)
```

Empty hash `sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=` is a **probe**. You run the build once, paste `got`, commit. Leaving the probe in git is how supply chain theatre happens.

Worked examples

Case 1: `fetchurl` with a probe hash

Save as `fod.nix`:

```
# fod.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.fetchurl {
  url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build fod.nix
```

Output (shape):

```
error: hash mismatch in fixed-output derivation '/nix/store/…-hello-2.12.1.tar.gz':
         specified: sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=
            got:    sha256-0P4s2aQwHzpwC3w3ys8ls6A6/JbuiJvTYdbBr9pQMnQ=
```

Paste `got` into `hash`, rebuild:

```
nix-build fod.nix
tar -tzf result | head
```

The path is now stable. Changing only the URL to a mirror of **the same bytes** keeps the path.

Case 2: `fetchFromGitHub`

Save as `gh.nix`:

```
# gh.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.fetchFromGitHub {
  owner = "NixOS";
  repo = "nix";
  rev = "2.35.2";
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

Use a **tag or full commit**. `rev = "master"` is a moving label; the hash will mismatch the moment master moves, or worse, you use `lib.fakeHash` and skip the mismatch.

Probe, paste, pin. Prefer `rev = "abc1234…"` (full git sha) for anything the desk ships.

`fetchFromGitHub` is `fetchzip` of the GitHub archive URL. `fetchgit` talks git and can set `leaveDotGit` (almost always **off** for `src`). `hash` is SRI; `sha256 = "0…"` still works and is the old base32 — new listings use `hash`. Do not set both.

Case 3: `nix-prefetch` so you do not fake-hash in a loop

```
nix-prefetch-url https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz
nix hash convert --hash-algo sha256 --to sri "$(nix-prefetch-url https://ftp.gnu.org/gnu/hel
```

or:

```
nix-prefetch-url --unpack https://github.com/NixOS/nix/archive/2.35.2.tar.gz
```

`fetchFromGitHub` unpacks; the hash is over the **unpacked** tree (`outputHashMode = "recursive"`), not the `.tar.gz` bytes. That is why a `fetchurl` hash and a `fetchFromGitHub` hash for "the same release" differ. `nix-prefetch-url` (packed) vs `nix-prefetch-url --unpack` / `nix-prefetch-github` (unpacked) — match the fetcher.

```
nix-prefetch-github NixOS nix --rev 2.35.2
```

`fetchpatch2` hashes the **applied** patch more reliably than `fetchpatch` (line endings, GitHub PR URLs). Prefer it on 26.05. `leaveDotGit = true` on `fetchgit` makes the hash include `.git` — usually wrong for `src`; use it only when the build runs `git describe`.

`fetchurl { name = "hello-2.12.1.tar.gz"; … }` sets the store basename; without `name`, Nix uses a hash of the URL and debugging `nix-store -q --references` is noisier.

Case 4: A package `src`

Save as `pkg.nix`:

```
# pkg.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.stdenv.mkDerivation rec {
  pname = "hello";
  version = "2.12.1";
  src = pkgs.fetchurl {
    url = "https://ftp.gnu.org/gnu/hello/hello-${version}.tar.gz";
    hash = "sha256-0P4s2aQwHzpwC3w3ys8ls6A6/JbuiJvTYdbBr9pQMnQ=";
  };
}
```

The `hash` in this listing is illustrative. Always use the `got` from **your** fetch. If GNU republishes the tarball, the mismatch is the news.

Case 5: `lib.fakeHash` is a development aid

```
hash = pkgs.lib.fakeHash;
```

It expands to a dummy SRI. Fine in a dirty worktree. `git grep fakeHash` must be empty on `main`.

The trap

The trap is **`sha256 = ""; or hash = lib.fakeHash` in production** so CI "stays green" when upstream retags. You are no longer pinning bytes. You are pinning a URL, which is not a pin.

The other trap is hashing the wrong thing: `fetchurl` hash on a GitHub **archive URL** while the package uses `fetchFromGitHub` (unpacked). Always mismatch. Prefetch with the same fetcher you will use.

A third: `rev = "v1.2.3"` on a repo that **moves tags**. Pin the git sha, put the tag in `version`.

The boring rule

- Every `src` is a FOD with a committed SRI `hash`.
- Probe once (`fakeHash` / `AAAA…`), paste `got`, commit.
- Pin `rev` to a git sha (tag in `version` only), never `master`.
- `git grep fakeHash` is empty on the default branch.
- Prefetch with the same fetcher (packed vs unpacked) you ship.
- `fetchpatch2` for remote patches. No `leaveDotGit` unless `git describe` is the build.

Try this

1. Run Case 1, paste the real hash, rebuild, then flip one character of the hash and read the mismatch.
2. `nix hash convert --help` and convert a hex sha256 to SRI.
3. Fetch the same hello tarball with `fetchurl` and, unpacked, with `fetchzip` / `fetchFromGitHub` if applicable. Confirm the hashes differ.
4. `git grep -n fakeHash` on a flake repo you maintain. Delete any hit on the main branch.

NIX_PATH, Channels, and Pinning Inputs

NIX_PATH, Channels, and Pinning Inputs

`<nixpkgs>` is whichever tree `$NIX_PATH` names today. The boring default is: **do not import from Nix PATH in anything you ship; pin with a flake lock on nixos-26.05, or `npins` if a repo cannot use flakes yet**.

This is the "input determinism" lesson: same expression, same bytes, six months later.

Mental model

Mechanism	What it pins	Use
`NIX_PATH` / `<nixpkgs>`	Whatever the machine has	Repl, throwaway `nix eval --impure`
Channels (`nix-channel`)	A moving URL per user	Legacy; do not start new work here
`npins` / `niv`	JSON of rev + hash in git	Non-flake trees
`flake.lock`	rev + `narHash`	**Desk default**

```
<nixpkgs>     →  lookup NIX_PATH  →  laptop A   laptop B
flake.lock    →  git-committed hash →  laptop A = laptop B = CI
```

`niv` still works. `npins` is the smaller, currently maintained pin tool. Flakes made both optional. `fetchTarball` **without** a hash is not a pin.

Worked examples

Case 1: See what `<nixpkgs>` actually is

```
echo "$NIX_PATH"
nix eval --impure --expr '<nixpkgs>'
```

Output (shape):

```
nixpkgs=/nix/var/nix/profiles/per-user/root/channels/nixpkgs:...
/nix/store/…-nixpkgs-unpacked
```

Two engineers with different channels evaluate different `hello`. `nix-channel --update` on a server is how prod drifts from CI.

```
nix-channel --list
```

If this prints `nixpkgs …` on a flake machine, that channel is still what `<nixpkgs>` sees. You do not need it. After the flake lock exists:

```
# optional: drop the unused channel so PATH lookups cannot surprise you
nix-channel --remove nixpkgs
```

Do not remove it until `flake.lock` is committed and `nix flake metadata` shows the 26.05 rev.

On NixOS the daemon's path is `nix.nixPath` / `nix.settings.nix-path`, not your user's leftover `~/.nix-defexpr`. A flake-first host can pin the lookup so even `--impure <nixpkgs>` is 26.05:

```
# nix-path.nix
{ pkgs, ... }:
{
  nix.nixPath = [ "nixpkgs=${pkgs.path}" ];
}
```

That is still **impure** for files that `import <nixpkgs>`. The boring production file uses `flake.lock`, not this. It only stops a surprise channel from winning `nix-shell -p`.

Case 2: Flake pin (the default)

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk pin";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    packages.x86_64-linux.hello = nixpkgs.legacyPackages.x86_64-linux.hello;
  };
}
```

```
nix flake lock
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.locked.rev'
```

Commit `flake.lock`. That rev is the pin. Branch `nixos-26.05` plus a lock hash — not `master`.

```
nix registry list
```

The **global** registry (`flake:nixpkgs -> github:NixOS/nixpkgs/nixpkgs-unstable`) is why unqualified `nixpkgs#hello` on a laptop is not 26.05. Pin it (`nix registry pin nixpkgs github:NixOS/nixpkgs/nixos-26.05`) or always type the URL / use `..` CI uses the repo lock, not

your registry. `nix.settings.flake-registry` can be set to an empty JSON on servers so they do not phone home for registry.json.

Case 3: `npins` without flakes

```
nix run github:andir/npins -- init
nix run github:andir/npins -- add github NixOS nixpkgs --branch nixos-26.05
```

Save as `default.nix`:

```
# default.nix
let
  sources = import ./npins;
  pkgs = import sources.nixpkgs { };
in
pkgs.hello
```

```
nix-build
```

`npins/sources.json` is the lockfile. Commit it. `niv` is the same idea with `nix/sources.nix` — do not mix niv and npins in one repo.

Case 4: Override NIX_PATH for one command (lab only)

```
NIX_PATH=nixpkgs=flake:nixpkgs nix eval --impure --expr 'import <nixpkgs> {}' --apply 'p: p.
```

Useful to compare. Not a team pin. The next shell forgets it.

Case 5: Fetch a tarball by hash (no flake, no npins)

Save as `pin.nix`:

```
# pin.nix
let
  nixpkgs = builtins.fetchTarball {
    url = "https://github.com/NixOS/nixpkgs/archive/nixos-26.05.tar.gz";
    sha256 = "0000000000000000000000000000000000000000000000000000";
  };
in
(import nixpkgs { }).hello
```

```
nix-build pin.nix
```

Probe the hash once (`lib.fakeHash` / the `got:` line). The tarball URL `nixos-26.05` **moves**; the `sha256` does not. When you want a newer 26.05 commit, bump **both**. `fetchTarball` without `sha256` is a channel with extra steps.

The trap

The trap is **`import <nixpkgs> {}` in `configuration.nix` on a server**. Rebuilds follow the last `nix-channel --update`. CI, the laptop, and prod disagree.

The other trap is pinning `master`. There is no "26.05" in the incident report. Branch `nixos-26.05` plus a lock hash.

A third: `fetchTarball` of `nixos-26.05.tar.gz` **without** `sha256` — GitHub serves a moving tarball. A fourth: mixing `npins` and `flake.lock` for the same nixpkgs.

The boring rule

- Ship flakes + `flake.lock` on 26.05.
- No `<nixpkgs>` in production files. Registry pin is for `nix shell nixpkgs#`, not for CI.
- `npins` only when flakes are forbidden. One pin tool per repo.
- `fetchTarball` without a hash is not a pin. Bump URL **and** hash together.

Try this

1. `echo $NIX_PATH` on two machines (or a container vs the host). Compare `<nixpkgs>` paths.
2. `nix flake metadata` and write down the nixpkgs `rev`.
3. `npins init` in a throwaway dir; open `sources.json`.
4. `git grep -n '<nixpkgs>'` on the desk repo — production files should be empty.
5. `nix registry list` and `nix flake metadata` — if they disagree, unqualified `nixpkgs#` is lying.

Building From Source

The Derivation Build Process

The Derivation Build Process

`stdenv.mkDerivation` is how Unix software becomes a store path. The boring default is: **keep the standard phases, override only the ones you must, and let `stdenv` set `$CC`, `$out`, RPATH, and the sandbox**.

Raw `derivation { }` (fundamentals) is the primitive. This chapter is what you write for real C.

Mental model

Phase	Default
`unpackPhase`	Unpack `$src`
`patchPhase`	`patch -p1` each of `patches`
`configurePhase`	`./configure --prefix=$out` if that script exists
`buildPhase`	`make`
`checkPhase`	`make check` if `doCheck = true`
`installPhase`	`make install`
`fixupPhase`	`patchelf` RPATH, strip, gzip man pages

Skip a phase with `dontConfigure = true;` (or `dontBuild`, `dontInstall` — rarely). Override a phase with `buildPhase = ''…'';`. Always `mkdir -p $out/bin` before `cp` if you write `installPhase` by hand.

`$CC` is the stdenv compiler (gcc or clang), already wrapped with hardening flags. Do not call `gcc` by name.

`nativeBuildInputs` run on the **build** machine (`make`, `pkg-config`, `$CC`). `buildInputs` are linked into the output (zlib, openssl). Mixing them is why a cross build links host openssl. `strictDeps = true;` makes that mix an eval/build error — turn it on for new C packages.

`enableParallelBuilding = true;` (and `enableParallelChecking`) is `make -j`. Default is often already on in 26.05 stdenv; if a Makefile is racy, set it false and say why.

Custom phases must still fire hooks:

```
buildPhase = ''
  runHook preBuild
  $CC -O2 -o desk-status-c main.c
  runHook postBuild
'';
```

Skipping `runHook` drops `preBuild` from `nativeBuildInputs` (code generators, `autoreconfHook`).

```
$src → unpack → patch → configure → build → check → install → fixup → $out
```

Worked examples

Case 1: One C file, no autotools

Save as `c_tool.nix`:

```
# c_tool.nix
{ lib, stdenv }:

stdenv.mkDerivation {
  pname = "desk-status-c";
  version = "1.0";
  src = ./.;
  dontConfigure = true;
  buildPhase = ''
    cat > main.c << 'EOF'
    #include <stdio.h>
    int main(void) {
      puts("Desk C worker active");
      return 0;
    }
    EOF
    $CC -O2 -o desk-status-c main.c
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp desk-status-c $out/bin/
  '';
  meta = {
    description = "Compiled C desk status check";
    license = lib.licenses.mit;
  };
}
```

Save as `default.nix`:

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./c_tool.nix { }
```

```
nix-build default.nix
./result/bin/desk-status-c
```

Output:

```
Desk C worker active
```

Case 2: `doCheck`

```
# in c_tool.nix
doCheck = true;
checkPhase = ''
  ./desk-status-c
'';
```

`checkPhase` runs **before** `installPhase`, in the build directory. The binary is still `./desk-status-c`, not `$out/bin`.

```
nix-build default.nix
```

A failing check fails the derivation. Good.

Case 3: `$src` as a real tree

Put `main.c` in git and drop the `cat` from `buildPhase`:

```
stdenv.mkDerivation {
  pname = "desk-status-c";
  version = "1.0";
  src = ./src;
  dontConfigure = true;
  buildPhase = ''
    $CC -O2 -o desk-status-c main.c
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp desk-status-c $out/bin/
  '';
}
```

Filter `src` (source-filtering chapter) so `.git` is not an input.

Case 4: Inspect the wrapped compiler

```
nix-build default.nix
nix-store -q --references result
```

You should see glibc (or musl), not gcc. gcc is a **build**-time input; fixupPhase did not copy it into the runtime closure.

```
readelf -d result/bin/desk-status-c | grep PATH
```

RUNPATH points at store libc. That is stdenv, not you.

Case 5: `outputs` (preview)

```
outputs = [ "out" "dev" ];
```

Multiple outputs split runtime (`out`) from headers (`dev`). Skip until a closure-size measurement says headers are the problem. One `$out` is the desk default.

The trap

The trap is **`installPhase` without `mkdir -p $out/bin`**. `cp: cannot create … No such file`. The build log is long; the last line is the one that matters.

The other trap is `gcc main.c` instead of `$CC`. Cross and clang stdenvs then break. `$CC` is the contract.

The boring rule

- Standard phases. `dontConfigure` when there is no `./configure`.
- `$CC`, not `gcc`. `$out`, not `/usr`. `nativeBuildInputs` vs `buildInputs`. `strictDeps` on new C.
- Custom phases call `runHook preBuild` / `postInstall`.
- `mkdir -p` before `cp` in a custom `installPhase`.
- `doCheck = true` when tests exist and are hermetic.
- Language builders (`buildGoModule`, `buildRustPackage`) wrap this. Use them when they exist.

Try this

1. Add Case 2; `nix-build`; then `checkPhase = "false";` and confirm failure.
2. Replace `$CC` with `gcc`; cross later will hurt — put `$CC` back.
3. `nix path-info -Shr result` vs `nix-store -q --deriver result` references — runtime vs build.
4. Forget `mkdir`; save the error; restore.

Building C and C++ Projects with CMake

Building C and C++ Projects with CMake

CMake projects should not grow a hand-written `cmake -B build` in `buildPhase`. The boring default is: **`cmake` in `nativeBuildInputs`, optional `ninja`, flags in `cmakeFlags`, and stdenv's configure/build/install hooks**.

Mental model

Putting `cmake` on `nativeBuildInputs` enables the CMake **setup hook**:

```
cmake -B build -DCMAKE_INSTALL_PREFIX=$out  [+ cmakeFlags]
cmake --build build
cmake --install build
```

`nativeBuildInputs` = tools that **run on the build machine** (cmake, ninja, pkg-config). `buildInputs` = libraries **linked into** the binary (zlib, openssl). Mixing them up is how cross breaks.

```
stdenv.mkDerivation {
  nativeBuildInputs = [ cmake ninja ];
  buildInputs = [ zlib ];
  cmakeFlags = [ "-DENABLE_DESK=ON" ];
}
```

Worked examples

Case 1: Tiny CMake project

Save as `cmake_project.nix`:

```nix
# cmake_project.nix
{ lib, stdenv, cmake }:

stdenv.mkDerivation {
  pname = "desk-math-engine";
  version = "1.0.0";
  src = ./.;
  nativeBuildInputs = [ cmake ];
```

```
  preConfigure = ''
    cat > CMakeLists.txt << 'EOF'
    cmake_minimum_required(VERSION 3.20)
    project(DeskMath C)
    add_executable(desk-calc main.c)
    install(TARGETS desk-calc DESTINATION bin)
    EOF
    cat > main.c << 'EOF'
    #include <stdio.h>
    int main(void) {
      puts("Desk Math Engine: 42");
      return 0;
    }
    EOF
  '';
  meta = {
    description = "Desk calculation engine";
    license = lib.licenses.mit;
  };
}
```

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./cmake_project.nix { }
```

```
nix-build default.nix
./result/bin/desk-calc
```

Output:

```
Desk Math Engine: 42
```

Move `CMakeLists.txt` / `main.c` into git and delete `preConfigure` for a real tree.

Case 2: `cmakeFlags`

```
cmakeFlags = [
  "-DCMAKE_BUILD_TYPE=Release"
  "-DBUILD_TESTING=OFF"
];
```

Do not append these onto a custom `buildPhase`. The hook reads `cmakeFlags`.

Case 3: Ninja

```
nativeBuildInputs = [ cmake ninja ];
```

The hook prefers Ninja when `ninja` is on `nativeBuildInputs`. Faster incremental compiles in the sandbox still start from zero each derivation — Ninja mainly helps huge trees and nicer logs.

Case 4: Link zlib

```
# cmake_zlib.nix
{ stdenv, cmake, zlib }:

stdenv.mkDerivation {
  pname = "desk-z";
  version = "1.0";
  src = ./.;
  nativeBuildInputs = [ cmake ];
  buildInputs = [ zlib ];
}
```

CMake's `find_package(ZLIB)` sees pkg-config / cmake files from the zlib derivation. You did not set `ZLIB_ROOT` by hand.

Add `pkg-config` when the project uses it:

```
nativeBuildInputs = [ cmake pkg-config ];
buildInputs = [ zlib openssl ];
```

`CMAKE_PREFIX_PATH` is already set by the setup hook. `strictDeps = true;` so a `buildInputs = [ cmake ];` mistake fails instead of linking host cmake into a cross build.

Case 5: Tests

```
doCheck = true;
# CMake: enable tests in cmakeFlags if needed
# cmakeFlags = [ "-DBUILD_TESTING=ON" ];
```

Hermetic tests only. A test that `curl`s the network fails the sandbox. `doCheck = false;` with a comment when upstream tests are broken on Nix — then a `passthru.tests` later.

The trap

The trap is `buildPhase = "cmake -B build && cmake --build build";`. You skip `CMAKE_INSTALL_PREFIX=$ou` and the stdenv flags. Use `cmakeFlags` and the hook.

The other trap is `buildInputs = [ cmake ];`. cmake then looks like a **runtime** library. `nativeBuildInputs`.

The boring rule

- `cmake` (and `ninja`) in `nativeBuildInputs`.
- Linked libs in `buildInputs`. `pkg-config` is native. `strictDeps` on new CMake packages.
- Options via `cmakeFlags`, not a custom cmake command.
- Real `CMakeLists.txt` in git; `preConfigure` only for the listing in this book.
- `doCheck` when tests do not need the network.

Try this

1. Case 2: add `-DCMAKE_BUILD_TYPE=Release`; `nix log` and grep `CMAKE_BUILD_TYPE`.
2. Move cmake to `buildInputs`; `nix-build`; note warnings or cross pain; put it back.
3. Add `zlib` and `find_package(ZLIB)` in a lab `CMakeLists.txt`.
4. `nix path-info -Shr result` — cmake should not appear in the **runtime** closure.

Rust Packages with Cargo

Rust Packages with Cargo

The sandbox cannot talk to crates.io. The boring default is: **`rustPlatform.buildRustPackage` with `cargoHash` (or `cargoLock.lockFile`) and a committed `Cargo.lock` — Crane is optional when that FOD is too coarse**.

This is the Rust peer of `buildGoModule`. `cargoSha256` is deprecated; use `cargoHash` (SRI).

Mental model

`Cargo.lock → cargo vendor FOD (cargoHash) → cargo build --offline --release → $out/bin`

Style	When
`cargoHash = "sha256-…"`	**26.05 default.** One hash of the vendored crates (`useFetchCargoVendor` is on).
`cargoLock.lockFile = ./Cargo.lock`	Git deps / patches in lockfile; `outputHashes` per git crate.
Crane	Incremental crate FODs (like gomod2nix). Extra flake input.
fenix / rust-overlay	Pin `rustc` newer than nixpkgs. Only when 26.05's `rustc` is too old.

Commit `Cargo.lock` even for bins (`cargo generate-lockfile`). Without it you are not reproducible. Filter `target/` out of `src`. `cargoSha256` is gone — `cargoHash` is SRI.

A Cargo **workspace** is `cargoRoot` / `buildAndTestSubdir`, not a second `src`. `strictDeps = true` when you have C libs (openssl): `pkg-config` stays native.

Worked examples

Case 1: `buildRustPackage` + `cargoHash`

Save as `Cargo.toml` (next to `src/main.rs` and `Cargo.lock`):

```
# Cargo.toml
[package]
name = "desk-rust-agent"
version = "0.1.0"
```

```
edition = "2021"
```

Save as `rust_app.nix`:

```
# rust_app.nix
{ lib, rustPlatform }:

rustPlatform.buildRustPackage {
  pname = "desk-rust-agent";
  version = "0.1.0";
  src = lib.cleanSource ./.;
  cargoHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  # 26.05 default is true; set only if a pin still uses the old vendor FOD.
  # useFetchCargoVendor = true;
  meta = {
    description = "Desk Rust agent";
    license = lib.licenses.mit;
    mainProgram = "desk-rust-agent";
  };
}
```

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./rust_app.nix { }
```

```
nix-build default.nix
./result/bin/desk-rust-agent
```

Probe `cargoHash` with `lib.fakeHash` once; paste `got:`. Never leave `fakeHash` on main. `edition = "2021"` or `"2024"` as the crate requires.

Case 2: `cargoLock.lockFile`

```
# lockfile.nix fragment
{
  src = ./.;
  cargoLock = {
    lockFile = ./Cargo.lock;
    outputHashes = {
      # only if the lockfile has git sources:
      # "some-git-crate-0.1.0" = "sha256-…";
    };
  };
}
```

Prefer this when the lockfile has `git` sources. Hash mismatches on git deps are FODs too — probe them. Do not set **both** `cargoHash` and `cargoLock` — pick one.

Workspace (one crate in a repo of several):

```
# workspace.nix fragment
{
  src = lib.cleanSource ./.;
  cargoRoot = ".";
  buildAndTestSubdir = "crates/desk-agent";
  cargoHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

`buildAndTestSubdir` is the package Cargo builds. The lockfile at `cargoRoot` still vendors **all** workspace crates that lockfile lists — keep `src` filtered, but include that `Cargo.lock`.

Case 3: Native libs (openssl, pkg-config)

```
# openssl-agent.nix
{ rustPlatform, pkg-config, openssl, lib }:

rustPlatform.buildRustPackage {
  pname = "desk-agent";
  version = "0.1.0";
  src = lib.cleanSource ./.;
  cargoHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  nativeBuildInputs = [ pkg-config ];
  buildInputs = [ openssl ];
  strictDeps = true;
}
```

`pkg-config` is native. `openssl` is linked. Same split as CMake. rustc should **not** appear in `nix path-info -Shr result`.

Case 4: Tests

`buildRustPackage` runs `cargo test` by default. Disable only with a reason:

```
{
  doCheck = true; # default: cargo test
  # cargoTestFlags = [ "--offline" ];
  # checkFlags = [ "--skip=needs_network" ];
  # doCheck = false; # only if upstream tests need network - see passthru.tests
}
```

Hermetic tests stay on. A test that hits crates.io or GitHub fails the sandbox — good. Skip **one** test with `checkFlags`; do not disable the suite to hide a missing `buildInputs`.

Case 5: Crane (optional)

Crane builds deps as their own derivations so a one-line `src` change does not rebuild the world of crates. Use it when CI of a large workspace is the pain. It is a flake input (`github:ipetkov/crane`) with `follows` on nixpkgs 26.05.

Do not start a 200-line `hello` crate on Crane. Measure `nix-build` first.

fenix / oxalica `rust-overlay`: pin a nightly or a rustc newer than 26.05. The desk default is **`pkgs.rustPlatform` from the channel**.

The trap

The trap is `cargoSha256`. Deprecated. `cargoHash` (SRI) or `cargoLock`, not both.

The other trap is no `Cargo.lock` in git because "it's a bin." Then every CI run resolves crates.io differently. Commit the lock.

`src = ./.;` including `target/` will bust hashes. `cleanSource` / fileset; add `/target` to `.gitignore`.

A third: `buildAndTestSubdir` without the workspace `Cargo.lock` in `src`. A fourth: rust-overlay "because nightly" for a crate that 26.05's `rustc` already builds.

The boring rule

- `buildRustPackage` + `cargoHash` + committed `Cargo.lock`. Not `cargoSha256`.
- Probe the hash; never leave `fakeHash` on main.
- Workspace: `buildAndTestSubdir` + lockfile at `cargoRoot`.
- `pkg-config` native; C libs in `buildInputs`; `strictDeps` when mixed.
- Crane / rust-overlay only when measured.
- Filter `target/` out of `src`. rustc not in the runtime closure.

Try this

1. Tiny crate; `fakeHash`; paste `got`; `nix-build`; run the bin.
2. Add a crates.io dep; bump `cargoHash`.
3. `doCheck = true` (default); add a `#[test]` that `assert!(true)`.
4. `nix path-info -Shr result` — rustc should not be in the **runtime** closure.
5. Add a git crate to `Cargo.toml`, switch Case 1 to `cargoLock` + `outputHashes`, probe that hash.

Python Packages with Pip and Setuptools

Python Packages with Pip and Setuptools

`pip install` on the server is a moving PyPI. The boring default is: **`python3.pkgs.buildPythonApplication` (CLI) or `buildPythonPackage` (library), deps from `python3.pkgs.*` on nixpkgs 26.05 — not pip in the sandbox**.

`poetry2nix` / `uv2nix` exist. Use them when a lockfile is the team contract **and** you have measured `buildPythonApplication` as too coarse. They are not the first recipe.

Mental model

Builder	For
`buildPythonApplication`	A CLI with `project.scripts` — wraps `PYTHONPATH`
`buildPythonPackage`	A library other Python code imports
`python3.withPackages (ps: [ ps.requests ])`	A **devShell** interpreter, not a shipped package

```
pyproject.toml
    build-system = setuptools / hatchling / …
    dependencies = python3.pkgs.requests  (from nixpkgs, not PyPI at build time)
```

```
$out/bin/desk-reporter   (wrapped)
```

`pyproject = true;` on modern trees. `format = "setuptools"` on old `setup.py` only — do not set both. The host python does not matter.

Attr (26.05)	Old gist name	Role
`build-system`	`nativeBuildInputs` (setuptools/hatchling)	Builds the wheel
`dependencies`	`propagatedBuildInputs`	Runtime Python deps
`nativeCheckInputs`	`checkInputs`	pytest and friends
`pythonImportsCheck`	(often omitted)	`import pkg` after install

`pytestCheckHook` in `nativeCheckInputs` is the nixpkgs default test runner. `pythonRelaxDeps` when an upstream pin is tighter than nixpkgs. Tests that hit the network fail in the sandbox — they must use `python3.pkgs`.

Worked examples

Case 1: Application

Save as `pyproject.toml`:

```
# pyproject.toml
[project]
name = "desk-reporter"
version = "1.0.0"
dependencies = ["requests"]
[project.scripts]
desk-reporter = "reporter:main"
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

Save as `desk_reporter.nix`:

```
# desk_reporter.nix
{ lib, python3 }:

python3.pkgs.buildPythonApplication {
  pname = "desk-reporter";
  version = "1.0.0";
  pyproject = true;
  src = lib.cleanSource ./.;
  build-system = [ python3.pkgs.setuptools ];
  dependencies = [ python3.pkgs.requests ];
  pythonImportsCheck = [ "reporter" ];
  meta = {
    description = "Desk metrics reporting CLI";
    license = lib.licenses.mit;
    mainProgram = "desk-reporter";
  };
}
```

```
# default.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.callPackage ./desk_reporter.nix { }
```

```
nix-build default.nix
./result/bin/desk-reporter
head -5 result/bin/desk-reporter
```

The wrapper's `PYTHONPATH` includes `requests`. Drop `requests` from `dependencies` and you get

`ModuleNotFoundError` — put it back. The `pyproject.toml` list is documentation; **nixpkgs attrs** are the pin.

Case 2: Library vs application

```
# desklib.nix
{ python3, lib }:

python3.pkgs.buildPythonPackage {
  pname = "desklib";
  version = "1.0.0";
  pyproject = true;
  src = lib.cleanSource ./.;
  build-system = [ python3.pkgs.setuptools ];
  dependencies = [ python3.pkgs.pydantic ];
}
```

Libraries do not always install a `bin/`. Applications do. Mixing them up gives a store path with no `bin/desk-reporter`.

Case 3: Tests

```
# tests fragment
{
  nativeCheckInputs = [
    python3.pkgs.pytestCheckHook
    python3.pkgs.requests-mock
  ];
  pythonImportsCheck = [ "reporter" ];
  disabledTests = [
    "test_talks_to_live_prometheus"
  ];
  doCheck = true;
}
```

`pytestCheckHook` is a **check** input, not `dependencies`, unless the app imports pytest at runtime (it should not). `pythonImportsCheck` catches a missing `dependencies` entry even when you skipped tests. `doCheck = false` is a last resort for a broken upstream suite — not a way to hide `pip install` in `checkPhase`.

`pythonRelaxDeps = [ "requests" ];` when `pyproject.toml` says `requests==2.31.0` and nixpkgs has 2.32. Prefer relaxing the pin over vendoring an old requests.

Case 4: Dev shell

```
# shell.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.mkShell {
  packages = [
    (pkgs.python3.withPackages (ps: [ ps.requests ps.pytest ]))
  ];
}
```

```
nix-shell --run "python -c 'import requests; print(requests.__version__)'"
which python
```

Must be a store path. Humans run `pytest` here. CI builds Case 1. Do not `pip install --user` inside the shell "just this once."

Case 5: Missing from nixpkgs

If `desk-internal-sdk` is not on 26.05:

1. `buildPythonPackage` it in *this* flake (`callPackage ./sdk.nix`).
2. Put it in `dependencies` of the app.

`uv2nix` / `poetry2nix` generate many package derivations from a lockfile. Worth it for an 80-dep web app. Not for a 40-line reporter. Overlaying a single `buildPythonPackage` is the boring missing-dep path.

The trap

The trap is **`pip install -r requirements.txt` in `buildPhase`**. Sandbox: no network. Even with `--offline`, those wheels are not Nix inputs unless you FOD them.

The other trap is `propagatedBuildInputs` copied from a 2019 gist. On current `buildPythonPackage`, use `dependencies` / `build-system` / `nativeCheckInputs` as in 26.05 nixpkgs python docs.

A third: **`format = "pyproject"` and `pyproject = true`** — pick `pyproject = true`. A fourth: putting `pytest` in `dependencies` so the CLI's runtime closure includes a test runner.

The boring rule

- `buildPythonApplication` for CLIs; `buildPythonPackage` for libs.
- `pyproject = true;` + `build-system` + `dependencies` from `python3.pkgs`. Not `format` + `pyproject`.
- No pip in the derivation. No unpinned `requirements.txt` as the only lock.

- Tests: `nativeCheckInputs` (`pytestCheckHook`), `pythonImportsCheck`, hermetic.
- `pythonRelaxDeps` for tight upstream pins. uv2nix/poetry2nix optional; one in-tree `buildPythonPackage` for a missing lib.

Try this

1. Case 1 with a real `pyproject.toml`; `nix-build`; `head -5 result/bin/desk-reporter` (wrapper).
2. Drop `requests` from `dependencies`; run; read `ModuleNotFoundError`; put it back.
3. `nix path-info -Shr result` and find `requests`.
4. `python3.withPackages` in a shell; `python -c 'import requests'`; confirm `which python` is a store path.
5. Drop `pythonImportsCheck`, misspell a dependency attr, watch the build succeed and the CLI fail. Put `pythonImportsCheck` back.

Go Modules with buildGoModule

Go Modules with buildGoModule

Go is how the desk API ships. The boring default is: **`pkgs.buildGoModule` with a real `vendorHash`, committed `go.mod` / `go.sum`, `CGO_ENABLED=0` unless you need C, and the Go from nixpkgs 26.05 — not `go get` in the sandbox**.

This is the Go analogue of `buildRustPackage`: one FOD for modules, one offline compile.

Mental model

```
go.mod + go.sum
       vendorHash (FOD)

/nix/store/…-desk-api-go-modules
       go build -mod=vendor  (offline)

$out/bin/desk-api
```

Field	Role
`vendorHash`	Hash of the vendored module tree. Probe with `lib.fakeHash`.
`vendorHash = null`	**Only** when there are **zero** module deps (stdlib only).
`proxyVendor`	Let the FOD use the module proxy (usual).
`modRoot`	If `go.mod` is not at `src` root.
`subPackages`	`["cmd/desk-api"]` so you do not build every `./...` main.
`env.CGO_ENABLED`	`"0"` for static-ish binaries; `"1"` needs gcc in `nativeBuildInputs`.

`go test` runs in `checkPhase` when `doCheck = true` (the default for `buildGoModule`). Tests must be hermetic.

Worked examples

Case 1: Stdlib-only binary

Save as `desk_service_go.nix`:

```nix
# desk_service_go.nix
{ lib, buildGoModule }:

buildGoModule {
  pname = "desk-service";
  version = "1.0.0";
  src = ./.;
  vendorHash = null;
  meta = {
    description = "Desk Go backend";
    license = lib.licenses.mit;
  };
}
```

With `go.mod` + `main.go` in `.` (no require lines):

```
nix-build -E 'with import <nixpkgs> {}; callPackage ./desk_service_go.nix {}'
./result/bin/desk-service
```

Output:

```
Desk Go backend operational
```

(Your `fmt.Println` text.)

Case 2: Dependencies — probe `vendorHash`

```nix
# desk_api.nix
{ buildGoModule, lib }:

buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = lib.fileset.toSource {
    root = ./.;
    fileset = lib.fileset.unions [ ./go.mod ./go.sum ./main.go ./internal ];
  };
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  subPackages = [ "." ];
}
```

```
nix-build
```

Mismatch prints `got: sha256-…`. Paste it. Changing `go.mod` without bumping the hash is a **stop**, not a network call in `buildPhase`.

Case 3: Pin the toolchain

```
# go-pin.nix
{ pkgs ? import <nixpkgs> { } }:

(pkgs.buildGoModule.override { go = pkgs.go_1_24; }) {
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
  vendorHash = null;
}
```

`pkgs.go` on 26.05 is whatever that channel ships. When the desk must match Boring-Go's 1.27 (or a security pin), override **the builder**, not `PATH` in a wrapper. Confirm with:

```
nix-build
./result/bin/desk-api
# or: strings result/bin/desk-api | grep '^go1\.'
```

Case 4: `CGO_ENABLED=0`

```
buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
  vendorHash = null;
  env.CGO_ENABLED = "0";
  ldflags = [ "-s" "-w" ];
}
```

No gcc. Cross to `GOOS=linux` is then just the Go toolchain. If you import `net` with cgo DNS, you want this anyway for boring deploys. SQLite with mattn/go-sqlite3 needs CGO **on** and `nativeBuildInputs = [ pkgs.pkg-config ];` plus the C library — then you left the boring path.

Case 5: Tests in the sandbox

```
buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
  vendorHash = "sha256-…";
  doCheck = true;
  # default checkPhase is go test
```

```
}
```

A test that dials `https://api.github.com` fails. Fixtures in `testdata/`. `vendorHash` must include test-only modules too (`go.sum` lines).

```
# go-api.nix fragment
{
  env.GOTOOLCHAIN = "local";
  env.CGO_ENABLED = "0";
  ldflags = [
    "-s" "-w"
    "-X main.version=${version}"
  ];
  tags = [ "netgo" ];
  proxyVendor = true; # FOD may use proxy.golang.org; buildPhase stays offline
}
```

`GOTOOLCHAIN=local` stops Go 1.21+ from downloading a newer toolchain because `go.mod` says `go 1.25`. The Nix `go` is the compiler. `proxyVendor = true` is the usual FOD; `false` expects an in-tree `vendor/`. Do not set both `vendorHash = null` and a populated `go.sum`.

The trap

The trap is **`vendorHash = null` with a `require` in `go.mod`**. Eval may succeed; the build tries the network and dies. `null` is not "figure it out."

The other trap is `go get` / `GOPROXY=https://proxy.golang.org` in `buildPhase`. The sandbox has no network. The FOD already fetched.

The boring rule

- `buildGoModule` + `vendorHash` (or `null` only for stdlib-only).
- Commit `go.mod` and `go.sum`. Filter `src` to them + packages.
- `subPackages` for the mains you ship.
- `CGO_ENABLED=0` unless a C library is the point. `GOTOOLCHAIN=local` on the derivation.
- Pin `go` via `buildGoModule.override { go = …; }` when the channel default is not the desk compiler.
- `ldflags` / `subPackages` / `tags` on the builder, not a `buildPhase` that calls `go build` by hand.

Try this

1. Stdlib `main.go`; `vendorHash = null`; `nix-build`; run the binary.
2. `go get github.com/google/uuid` in a throwaway module; set `fakeHash`; paste `got`.

3. Flip one character of `vendorHash`; read the mismatch.
4. `env.CGO_ENABLED = "0"`; `file result/bin/desk-api` — dynamically linked to musl/glibc still possible via Go's net; the point is no gcc.

Cross-Compilation for Multiple Platforms

Cross-Compilation for Multiple Platforms

Hand-rolled `aarch64-linux-gnu-gcc` prefixes are a wiki. The boring default is: **`pkgs.pkgsCross.<target>` on nixpkgs 26.05, or a native remote builder when you need speed and KVM**.

Mental model

Platform	Meaning
`build`	Machine compiling (laptop)
`host`	Machine running the output
`target`	What a compiler emits (only when building gcc)

`pkgsCross.aarch64-multiplatform.hello` is hello for ARM, built on x86_64 with a cross toolchain.

Remote builder (`ssh-ng` to an ARM box) is **native**. Prefer it for Go, kernels, and NixOS tests. Cross is fine for small C. Go also has `GOOS`/`GOARCH` (its own chapter) — that is not `pkgsCross`.

```
x86_64 laptop
    pkgsCross.aarch64-multiplatform.stdenv.cc


    ELF aarch64  (cannot run here without qemu-user)
```

Worked examples

Case 1: Cross hello

```
nix build -f '<nixpkgs>' pkgsCross.aarch64-multiplatform.hello
file result/bin/hello
```

Output (shape):

```
result/bin/hello: ELF 64-bit LSB executable, ARM aarch64, …
```

You cannot run it on x86_64 without qemu-user. `file` first, then decide.

Case 2: Your C program

Save as `cross_demo.nix`:

```
# cross_demo.nix
let
  pkgs = import <nixpkgs> { };
  armPkgs = pkgs.pkgsCross.aarch64-multiplatform;
in
armPkgs.stdenv.mkDerivation {
  pname = "desk-arm";
  version = "1.0";
  src = pkgs.writeTextDir "main.c" ''
    #include <stdio.h>
    int main(void) {
      puts("desk-arm");
      return 0;
    }
  '';
  buildPhase = ''
    $CC $src/main.c -o desk-arm
  '';
  installPhase = ''
    mkdir -p $out/bin
    cp desk-arm $out/bin/
  '';
}
```

```
nix-build cross_demo.nix
file result/bin/desk-arm
```

`$CC` is the cross compiler. Do not hardcode `gcc`. The sandbox's `gcc` would emit x86_64.

Case 3: nativeBuildInputs vs buildInputs

Save as `zlib-arm.nix` fragment inside a derivation:

```
# zlib-arm.nix
{ stdenv, pkg-config, zlib }:

stdenv.mkDerivation {
  pname = "desk-z";
  version = "1.0";
  src = ./.;
  nativeBuildInputs = [ pkg-config ]; # runs on the laptop
  buildInputs = [ zlib ];             # links into the ARM binary
```

```
  doCheck = stdenv.buildPlatform == stdenv.hostPlatform;
}
```

Tests that **execute** the binary must be off when crossing (`doCheck` false). `pkg-config` is a build tool; `zlib` is a host library. Swap them and the ARM binary links x86 zlib — or eval fails.

Case 4: Flake output for the ARM artifact

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk ARM binary";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      packages.x86_64-linux.desk-arm =
        pkgs.pkgsCross.aarch64-multiplatform.callPackage ./cross_demo.nix { };
    };
}
```

```
nix build .#desk-arm
file result/bin/desk-arm
```

The package lives under `packages.x86_64-linux` because that is **where it was built**. The ELF is still aarch64. Do not confuse flake `system` with `file` output.

Case 5: Prefer a builder for NixOS

`nixos-rebuild` / `nixosTest` for aarch64: remote `aarch64-linux` builder with `kvm`. Cross-compiling a full NixOS closure is possible and painful.

```
nix build .#packages.aarch64-linux.desk-api --max-jobs 0
```

`--max-jobs 0` forces the remote builder. If you have no builder, that error is correct — do not reach for `qemu-binfmt` as the release path.

The trap

The trap is **running `./result/bin/desk-arm` on the laptop** and calling the compiler broken. `file` first. `qemu-user` if you must execute; a builder if you must test for real. The other trap is `doCheck = true` on a cross drv that execs the binary.

The boring rule

- `pkgsCross.*` for small C. Native builder for OS and Go.
- `$CC`, not `gcc`. `nativeBuildInputs` vs `buildInputs`.
- `doCheck` only when build == host.
- 26.05 pin so the cross toolchain matches prod.
- `file` the output. Flake `system` is the builder, not the ELF.

Try this

1. `nix eval --raw -f '<nixpkgs>' 'pkgsCross.aarch64-multiplatform.stdenv.cc' | xargs basename`
2. Case 1 `file`.
3. Enable `doCheck = true` on a cross drv that runs the binary; watch it fail; add the platform guard.
4. `nix eval --expr 'builtins.attrNames (import <nixpkgs> {}).pkgsCross'` and pick one target you actually own.

Trivial Builders

Trivial Builders

Not every package needs `stdenv.mkDerivation` and a compiler. The boring default is: **`writeShellApplication` / `writeText` / `runCommand` / `symlinkJoin` for scripts, files, and combining closures — reach for `mkDerivation` when you compile**.

Mental model

nixpkgs "trivial builders" are functions that produce derivations without a full stdenv phase dance.

Builder	Produces
`writeText` / `writeTextFile`	One file at `$out`
`writeTextDir`	`$out/name`
`writeScript` / `writeScriptBin`	Executable (weak shebang)
`writeShellApplication`	Script + `runtimeInputs` + `set -euo pipefail`
`runCommand`	Tiny custom builder
`symlinkJoin`	One dir of symlinks to many packages
`copyPathToStore`	Copy a path as a store object

`writeShellApplication` is the desk default for shell. `writeScriptBin` is the older, sloppier cousin.

Worked examples

Case 1: `writeText`

Save as `banner.nix`:

```
# banner.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.writeText "desk-banner.txt" ''
  desk operations - window A
''
```

```
nix-build banner.nix
cat result
```

Output:

```
desk operations - window A
```

`$out` *is* the file. No `bin/`.

Case 2: `writeShellApplication`

Save as `tool.nix`:

```
# tool.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.writeShellApplication {
  name = "desk-status";
  runtimeInputs = [ pkgs.coreutils pkgs.jq ];
  text = ''
    echo '{"window":"A","ok":true}' | jq -c .
  '';
}
```

```
nix-build tool.nix
./result/bin/desk-status
```

Output:

```
{"ok":true,"window":"A"}
```

`jq` is on PATH inside the wrapper only.

Case 3: `runCommand`

Save as `run.nix`:

```
# run.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.runCommand "desk-tree" { } ''
  mkdir -p $out/share/desk
  echo 1.0 > $out/share/desk/version
  ln -s ${pkgs.hello}/bin/hello $out/hello-link
''
```

```
nix-build run.nix
cat result/share/desk/version
```

Use `runCommand` when you need a few `mkdir`/`cp` lines. If it grows a `configurePhase`, you wanted `mkDerivation`.

Case 4: `symlinkJoin`

Save as `join.nix`:

```
# join.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.symlinkJoin {
  name = "desk-cli";
  paths = [ pkgs.jq pkgs.ripgrep pkgs.hello ];
}
```

```
nix-build join.nix
ls result/bin
```

One GC root, several tools. Home Manager `home.packages` is this idea at user scale. A "kitchen sink" join of gcc+python+node is how closures explode — join what operators actually run.

Case 5: `writeTextDir` for a default config

```
# conf.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.writeTextDir "desk/config.json" (builtins.toJSON {
  window = "A";
  replicas = 2;
})
```

```
nix-build conf.nix
cat result/desk/config.json
```

Good for embedding JSON next to a service. Not for secrets.

The trap

The trap is **`writeScriptBin` with `curl` in the script** and no `runtimeInputs`. It works on your PATH and dies in systemd. `writeShellApplication` + `runtimeInputs`.

The other trap is `runCommand` that compiles C. That belongs in `stdenv`. Trivial builders are trivial.

The boring rule

- Scripts: `writeShellApplication`.
- One file: `writeText` / `writeTextDir`.
- Glue: `runCommand`. Union of bins: `symlinkJoin`.
- Compilers: `mkDerivation` / language builders.
- No secrets in `writeText`.

Try this

1. Case 2: add `pkgs.hello` to `runtimeInputs` and call `hello`.
2. `symlinkJoin` jq+hello; `nix path-info -Shr result | tail -1`.
3. Replace `writeShellApplication` with `writeScriptBin` and drop jq from the host PATH in a clean `nix shell --pure`; watch it fail; restore.
4. `writeText` a unit snippet and `cat` it; confirm `$out` is a file, not a directory.

Source Filtering

Source Filtering

`src = ./.` hashes the README, `.git`, and `result`. The boring default is: **`lib.fileset` (or `cleanSourceWith`) so only the files the build reads are in the FOD**, especially in a monorepo.

Mental model

The source tarball is an **input**. Extra files → extra hash → extra rebuilds → extra cache misses.

Tool	When
`lib.cleanSource`	Drop `.git`, `result`, editor junk. Fast default.
`lib.cleanSourceWith`	Plus a custom `filter`
`lib.fileset.toSource`	Union of paths; **preferred** on 26.05
`nix-gitignore.gitignoreSource`	Follow `.gitignore`

```
repo/
  apps/desk-api/   ← fileset
  docs/            ← not in src
  README.md        ← not in src
```

Never raw `src = ./.;` on a repo root for a leaf package. Include lockfiles (`go.sum`, `Cargo.lock`, `package-lock.json`). Exclude docs and `.git`.

Worked examples

Case 1: `cleanSource`

Save as `filter.nix`:

```nix
# filter.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.stdenv.mkDerivation {
  pname = "desk-src-demo";
  version = "1.0";
  src = pkgs.lib.cleanSource ./.;
  buildPhase = "echo ok > $out";
```

```
  installPhase = "true";
}
```

```
nix-build filter.nix
echo "# noise" >> README.md
nix-build filter.nix --dry-run
```

If README is filtered, the second command prints an empty "will be built" list. If gcc appears, README is still in `src`.

Case 2: `fileset` union

Save as `fileset.nix`:

```
# fileset.nix
{ pkgs ? import <nixpkgs> { }, lib ? pkgs.lib }:

pkgs.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = lib.fileset.toSource {
    root = ./.;
    fileset = lib.fileset.unions [
      ./main.go
      ./go.mod
      ./go.sum
    ];
  };
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

A CSS change under `./web` does not rebuild this derivation. Forgetting `go.sum` makes `go` complain — add it. fileset is stricter than gitignore.

`lib.fileset.gitTracked ./.` is "what git would ship," then subtract:

```
# fileset-git.nix fragment
src = lib.fileset.toSource {
  root = ./.;
  fileset = lib.fileset.difference
    (lib.fileset.gitTracked ./.)
    (lib.fileset.unions [ ./docs ./target ./.github ]);
};
```

`gitTracked` needs a `.git` at eval — fine in the repo, **not** inside a `fetchFromGitHub` unpack that has no `.git`. For a fetched `src`, union the files you need; do not call `gitTracked`.

Case 3: Exclude by name

```
# exclude.nix fragment
{
  src = pkgs.lib.cleanSourceWith {
    src = ./.;
    filter = path: type:
      let base = baseNameOf path; in
      !(builtins.elem base [ "README.md" "docs" ".envrc" "result" ]);
  };
}
```

The filter itself must be deterministic. Do not `readDir` a changing cache dir. Do not use `builtins.currentTime`.

Case 4: gitignore

```
# gitignore.nix fragment
{
  src = pkgs.nix-gitignore.gitignoreSource [ ] ./.;
}
```

Only as good as `.gitignore`. If you commit `data.bin` that the build does not need, it still busts. Prefer `fileset` when you can name the files. One style per repo — not both fighting.

Case 5: Prove it with `--dry-run`

```
nix-build fileset.nix
echo x >> README.md
nix-build fileset.nix --dry-run
echo x >> main.go
nix-build fileset.nix --dry-run
```

README edit: empty will-build list. `main.go` edit: the Go derivation rebuilds. That is the filter working.

```
nix path-info $(nix-build fileset.nix --no-out-link --argstr ignore 0 2>/dev/null; nix-insta
```

Or inspect the src path from `nix show-derivation` and confirm `.git` is absent.

```
drv=$(nix-instantiate fileset.nix)
nix show-derivation "$drv" | jq -r '.[].inputSrcs[]'
```

No `.git`, no `target/`, no `node_modules`. If `README.md` is in `inputSrcs`, the fileset still includes it — tighten the union.

The trap

The trap is **filtering out `go.sum` / `Cargo.lock`**. The FOD hash of vendors depends on the lock. Include lockfiles. Exclude docs and `.git`.

The other trap is `filterSource` that depends on `builtins.currentTime` or `readDir` of a changing cache dir. The filter itself must be deterministic.

A third: `gitTracked` on a `fetchFromGitHub` tree (no `.git`). A fourth: mixing `gitignoreSource` and `fileset` so nobody knows which filter won.

The boring rule

- Never raw `src = ./.` on a repo root for a leaf package.
- `fileset` on 26.05 when you can name the files. `difference` + `gitTracked` in a git checkout.
- Lockfiles in; READMEs, `target/`, `.git` out.
- `--dry-run` after a docs-only edit.
- One filter style per repo (`fileset` *or* gitignore, not both fighting).

Try this

1. Case 1 with and without `cleanSource`; `--dry-run` after a README edit.
2. `fileset` of a single `main.go`; add a sibling file; `--dry-run`.
3. Forget `go.sum` in the fileset; watch `go` complain; add it.
4. `nix show-derivation` the src input; confirm `.git` is absent.

Patching Packages

Patching Packages

Upstream is almost right. The boring default is: **a `.patch` file in git, `patches = [ ./desk-fix.patch ];` or `prev.pkg.overrideAttrs`, not a fork of nixpkgs**.

Mental model

`stdenv` `patchPhase` runs `patch -p1` for each entry in `patches`. Fetch a patch with `fetchpatch` / `fetchpatch2` when it lives on GitHub; vendor it in-tree when it is *your* fix.

```
src tarball  →  patchPhase  →  configure/build

                    ./desk-fix.patch  (in git)
```

Overlays compose: `prev.hello.overrideAttrs (old: { patches = (old.patches or []) ++ [ ./x.patch ]; })`.

`fetchpatch2` is the 24.11+ spelling that hashes the **applied** result more reliably. On 26.05, prefer `fetchpatch2` for remote patches. A patch is an input: change the diff → new hash → rebuild.

Worked examples

Case 1: In-tree patch on your package

Save as `desk-fix.patch` (illustrative unified diff):

```
# desk-fix.patch - unified diff against src root
--- a/main.c
+++ b/main.c
@@ -1,3 +1,3 @@
 int main(void) {
-  return 1;
+  return 0;
 }
```

Save as `patched.nix`:

```nix
# patched.nix
{ stdenv }:

stdenv.mkDerivation {
  pname = "desk-c";
  version = "1.0";
  src = ./src;
  patches = [ ./desk-fix.patch ];
}
```

```
nix-build -E 'with import <nixpkgs> {}; callPackage ./patched.nix {}'
```

The patch is an input. Change the diff → new drv. That is what you want. Generate with `git diff` from the unpacked `src` root so `-p1` matches.

`patchFlags = [ "-p1" ];` is the stdenv default. A diff generated from inside a subdirectory needs `-p2` — set `patchFlags`, do not hand-edit the hunk headers unless you must. `chmod +x` after a patch that adds a script: `postPatch`.

Case 2: Overlay on nixpkgs hello

Save as `overlay.nix`:

```nix
# overlay.nix
final: prev: {
  hello = prev.hello.overrideAttrs (old: {
    patches = (old.patches or [ ]) ++ [ ./hello-desk.patch ];
  });
}
```

You do not copy hello's expression. You append. `final.hello.overrideAttrs` here is infinite recursion — `prev`.

If upstream already has `patches`, `(old.patches or [ ]) ++` is required. Replacing `patches = [ ./x.patch ];` **drops** nixpkgs's patches and is a silent CVE.

Case 3: `fetchpatch2` from a PR

Save as `fetch-patch.nix`:

```nix
# fetch-patch.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.hello.overrideAttrs (old: {
  patches = (old.patches or [ ]) ++ [
    (pkgs.fetchpatch2 {
```

```
      url = "https://github.com/example/hello/commit/abc123.patch";
      hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
      # decode = "unzip";  # only if the URL is a zip of a patch
    })
  ];
})
```

```
nix-build fetch-patch.nix
```

Probe the hash. When GitHub force-pushes the PR, the hash mismatches — good. Pin a **commit** URL (`/commit/<full-sha>.patch`), not `/pull/1.patch` which moves. `fetchpatch` (v1) is the old hasher; `fetchpatch2` on 26.05.

GitHub may wrap the patch (CRLF, subject prefix). If `patchPhase` fails after a good hash, `fetchpatch2` still did its job — the **hunk** does not apply to this `src`. Rebase the patch against 26.05's tarball, do not `patchFlags = [ "-p0" "--force" ]`.

Case 4: `sed` in `postPatch` (last resort)

```
# postpatch.nix fragment
{
  postPatch = ''
    substituteInPlace Makefile --replace-fail /usr/local "$out"
  '';
}
```

`--replace-fail` (not `--replace`, which was removed) so a silent no-op cannot ship. `--replace-warn` is for a transition. Prefer a real patch when the change is more than one substitution. `substituteInPlace` runs in `postPatch` **after** `patches` — order matters if the hunk and the sed touch the same line.

Case 5: Drop an upstream patch

```
# drop.nix
final: prev:
let
  pkgs = prev;
in
{
  hello = prev.hello.overrideAttrs (old: {
    patches = builtins.filter
      (p: !(pkgs.lib.hasSuffix "bad.patch" (baseNameOf (toString p))))
      (old.patches or [ ]);
  });
}
```

Rare. Document *why* in a comment. Next nixpkgs bump may make the filter a no-op — check the diff in the lock PR.

The trap

The trap is **editing files in `/nix/store/...-hello` "to test."** Read-only. Copy the expression, patch, overlay.

The other trap is a patch with Windows line endings or the wrong `-p` strip level. `patchPhase` fails. Generate with `git diff` from the unpacked `src` root.

A third: `patches = [ ./x.patch ];` wiping nixpkgs's list. A fourth: `/pull/N.patch` as a FOD URL.

The boring rule

- Your fix: `./foo.patch` in git + `patches = (old.patches or []) ++ [ ]`.
- Upstream PR: `fetchpatch2` + commit URL + hash. Not `/pull/1.patch`.
- Overlay `overrideAttrs` from `prev`; do not fork nixpkgs for one hunk.
- `substituteInPlace --replace-fail` for path rewrites, after patches.
- Patch is an input. It must apply cleanly on 26.05's `src`.

Try this

1. `git diff` a one-line change; save as `.patch`; add to `patches`; `nix-build`.
2. Break the hunk context; read `patchPhase` failure in `nix log`.
3. Overlay `hello` with an extra empty patch file and `nix-build` under that overlay.
4. `fetchpatch2` with `fakeHash`, paste `got`.

Node.js Packages

Node.js Packages

Node is the desk frontend. The boring default is: **`buildNpmPackage` + `npmDepsHash` from `package-lock.json`, or `pnpm.fetchDeps` when the lockfile is pnpm — not `npm install` in `buildPhase`.**

Go moved to its own chapters. This page is Node only.

Mental model

Same two-step FOD as Go:

```
package-lock.json  →  npmDepsHash FOD  →  npm ci --offline  →  $out
```

Lock	Builder
`package-lock.json`	`buildNpmPackage`
`pnpm-lock.yaml`	`pnpm.fetchDeps` + `pnpm` hooks (26.05)
Yarn	yarn-specific fetchers; do not mix

Commit the lockfile. `npm install` without a lock is not a pin. `npm install` in `buildPhase` hits the sandbox (`EAI_AGAIN`).

Worked examples

Case 1: `buildNpmPackage`

Save as `node_app.nix`:

```nix
# node_app.nix
{ buildNpmPackage, lib }:

buildNpmPackage {
  pname = "desk-web";
  version = "1.0.0";
  src = lib.cleanSource ./.;
  npmDepsHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  installPhase = ''
```

```
    runHook preInstall
    mkdir -p $out/dist
    cp -r dist/* $out/dist/
    runHook postInstall
  '';
}
```

```
nix-build -E 'with import <nixpkgs> {}; callPackage ./node_app.nix {}'
```

First run prints `got: sha256-…`. Paste it into `npmDepsHash`. `npm ci` uses the lock, not the latest registry. Never leave `lib.fakeHash` on `main`.

Case 2: A CLI with a bin

If `package.json` has `"bin": { "desk-web": "bin/cli.js" }`, `buildNpmPackage` often wraps it:

```
ls result/bin
./result/bin/desk-web --help
```

If you only ship static `dist/` for nginx, you do **not** want `nodejs` in the runtime closure. Check:

```
nix path-info -Shr ./result | grep nodejs || echo 'node not in runtime (good for static dist
```

Case 3: pnpm

Save as `pnpm.nix`:

```
# pnpm.nix
{ lib, stdenv, pnpm, nodejs }:

stdenv.mkDerivation (finalAttrs: {
  pname = "desk-web";
  version = "1.0.0";
  src = lib.cleanSource ./.;
  pnpmDeps = pnpm.fetchDeps {
    inherit (finalAttrs) pname version src;
    hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
    fetcherVersion = 2;
  };
  nativeBuildInputs = [ nodejs pnpm.configHook ];
  buildPhase = ''
    pnpm run build
  '';
  installPhase = ''
```

```
    mkdir -p $out
    cp -r dist $out/
  '';
})
```

`fetcherVersion` and option names move; the 26.05 nixpkgs doc for `pnpm.fetchDeps` wins if this snippet drifts. Do not mix npm lock + pnpm fetcher.

Case 4: Node in a devShell, not in the output

Save as `flake.nix`:

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      packages.x86_64-linux.desk-web = pkgs.callPackage ./node_app.nix { };
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [ pkgs.nodejs pkgs.pnpm ];
      };
    };
}
```

```
nix develop --command node --version
```

CI builds with Case 1. Humans `pnpm test` inside `nix develop`. Do not `npm install -g` on the laptop.

Case 5: Native addons

`node-gyp` needs `python3`, `gcc`, headers:

```
# addon.nix fragment
{
  nativeBuildInputs = [ pkgs.python3 pkgs.pkg-config pkgs.nodejs ];
  buildInputs = [ pkgs.openssl ];
}
```

If the addon can be skipped (`optionalDependencies`), prefer a pure-JS path for the desk image. Native addons are a CGO-shaped exception — document gcc in the package comment.

```
# node_app.nix fragment
{
  npmBuildScript = "build"; # package.json script name; default is often "build"
  makeCacheWritable = true; # some postinstall scripts write into node_modules
  forceGitDeps = true;      # only if the lock has git: URLs
}
```

`NODE_ENV=production` in `installPhase` is how you drop devDependencies from the **runtime** tree. A static `dist/` served by nginx should not contain `node_modules` at all — `cp -r dist` only. `nodejs` in `nativeBuildInputs` (build) vs not in the runtime closure is the same split as gcc vs glibc.

The trap

The trap is **`npm install` in `buildPhase`**. Sandbox: `EAI_AGAIN` / network disabled. Hash the lockfile FOD.

The other trap is no lockfile (`package-lock.json` gitignored). Then `npmDepsHash` is a lie next week.

The boring rule

- Lockfile in git. `npmDepsHash` / `pnpm.fetchDeps` hash.
- `buildNpmPackage` for npm. pnpm fetcher for pnpm. Do not mix.
- Probe hashes; never leave `fakeHash` on main.
- Toolchain in the devShell; artifacts from the derivation.
- Native addons are a CGO-shaped exception — document gcc.
- Static `dist/` should not keep `nodejs` in `nix path-info -Shr`. `npmBuildScript` / `makeCacheWritable` only when the log says you need them.

Try this

1. Tiny `package.json` + lock; `fakeHash`; paste `got`.
2. Change a dep in `package.json` without `npm install`; watch the FOD mismatch or `npm ci` fail.
3. `nix path-info -Shr result` on a built `dist`; confirm `nodejs` is or is not in the runtime closure (static `dist/` should not need node if nginx serves files).
4. `git check-ignore -v package-lock.json` — it must **not** be ignored.

Go Toolchains and gomod2nix

Go Toolchains and gomod2nix

`buildGoModule` is Cargo's `buildRustPackage`. **gomod2nix** is closer to **Crane**: it generates Nix from `go.mod` so module FODs are per-crate, not one giant `vendorHash`. The boring default is still **`buildGoModule`** until a repo's vendor FOD is painfully slow; then gomod2nix.

Pin the compiler with `buildGoModule.override { go = pkgs.go_1_24; }` or an overlay. `go` in nixpkgs 26.05 *is* the toolchain. dream2nix analogues exist and are **not** the desk default.

Mental model

Tool	Like (Rust)	When
`buildGoModule` + `vendorHash`	`buildRustPackage` + `cargoHash`	**Default**
`gomod2nix`	Crane	Many modules, incremental FODs
`go` overlay / `.override { go = … }`	fenix / rust-overlay	Pin compiler version
`vendor/` in git	`cargo vendor` in-tree	Air-gap; last resort

```
go.mod   gomod2nix    gomod2nix.toml + buildGoApplication
          buildGoModule    one vendorHash FOD
```

One compiler version per flake. CI, laptop, and `buildGoModule` share it.

Worked examples

Case 1: Stay on `buildGoModule`

If `nix-build` of the API is seconds after a cache hit, stop. gomod2nix is extra files to review.

```nix
# desk-api.nix
{ pkgs }:

pkgs.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
```

```
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

Probe `vendorHash`. Never leave `lib.fakeHash` on main.

Case 2: Pin Go like fenix pins rustc

Save as `overlay.nix`:

```
# overlay.nix
final: prev: {
  deskGo = prev.go_1_24;
  buildGoModule = prev.buildGoModule.override { go = final.deskGo; };
}
```

Then `pkgs.buildGoModule { … }` in that `pkgs` uses 1.24. Document the version in the flake description. When 26.05's default `go` is already what you want, **do not** overlay.

```
nix eval -f '<nixpkgs>' go.version
```

Write that version down. Bump it with the channel, not with `go install`.

Case 3: gomod2nix generate

```
nix run github:nix-community/gomod2nix -- --dir .
```

That writes `gomod2nix.toml`. Commit it.

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-api via gomod2nix (optional)";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    gomod2nix = {
      url = "github:nix-community/gomod2nix";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, gomod2nix }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
```

```
    in
    {
      packages.${system}.desk-api =
        gomod2nix.legacyPackages.${system}.buildGoApplication {
          pname = "desk-api";
          version = "1.0.0";
          src = ./.;
          modules = ./gomod2nix.toml;
        };
    };
}
```

Exact attr names (`buildGoApplication` vs `mk`) follow the gomod2nix README for the lock you pin. `follows` so you do not eval two nixpkgs.

Case 4: `vendor/` in git (air-gap)

```
go mod vendor
```

```
# vendor.nix fragment
{
  src = ./.;
  vendorHash = null;
}
```

On current `buildGoModule`, an in-tree `vendor/` plus the right flags skips the FOD. It bloats git. Use when the module proxy is forbidden. Prefer `vendorHash` FODs on the desk.

Case 5: Private modules

The vendor FOD **may** use the network. Give Nix `access-tokens` in `nix.conf` (CI secrets), not a password in the expression.

```
# do not
{
  GOPROXY = "https://user:hunter2@git.desk.internal";
}
```

```
# /etc/nix/nix.conf  (CI runner / ops)
access-tokens = git.desk.internal=…
```

`GOPRIVATE=git.desk.internal` belongs in the **devShell**, not as a store string that leaks into `nix why-depends`.

The trap

The trap is **switching to gomod2nix because a blog compared it to Crane**, then checking in 400 generated files for a 3-module API. Measure `nix-build` time first.

The other trap is three Go versions: `home.packages = [ go_1_22 ]`, a devShell `go_1_24`, and `buildGoModule` default. One overlay / one `override`. CI and laptop match.

The boring rule

- `buildGoModule` first. gomod2nix when vendor FODs hurt.
- Pin `go` with `.override { go = pkgs.go_1_24; }` (or whatever 26.05 offers), not a random tarball.
- `follows` on gomod2nix's nixpkgs.
- No dream2nix as the desk default.
- One compiler version per flake. Tokens in `nix.conf`, not in Nix strings.

Try this

1. `nix eval -f '<nixpkgs>' go.version` on 26.05. Write it down.
2. Overlay `buildGoModule` to `go_1_24` if that attr exists; `strings` the binary for `go1.`.
3. `nix run github:nix-community/gomod2nix -- --help`.
4. `git grep GOPATH` in the flake — should not set a host GOPATH for the build.

Go Development Environments

Go Development Environments

A Go service that builds in CI and not in `$HOME/sdk` is the point of Nix. The boring default is: **`mkShell` with `go`, `gopls`, `gotools`, and `golangci-lint` from the same 26.05 pin as `buildGoModule` — direnv loads it; Home Manager does not install Go.**

Mental model

```
flake.nix
  packages.desk-api      = buildGoModule { … }
  devShells.default      = mkShell { packages = [ go gopls … ]; }
```

Same `go` attr in both. `direnv` + `use flake`. `CGO_ENABLED=0` in the shell if the package sets it.

Private modules: `GOPRIVATE=git.desk.internal` in the shell. Tokens stay in the environment / netrc, not in the flake.

`GOTOOLCHAIN=local` stops Go 1.21+ from downloading a different toolchain from `go.mod`'s `go` line. The Nix `go` is the compiler.

Worked examples

Case 1: Shell that matches the package

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-api shell";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
      go = pkgs.go;
    in
    {
      packages.${system}.desk-api = pkgs.buildGoModule {
```

```
        pname = "desk-api";
        version = "1.0.0";
        src = ./.;
        vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
      };

      devShells.${system}.default = pkgs.mkShell {
        packages = [
          go
          pkgs.gopls
          pkgs.gotools
          pkgs.golangci-lint
          pkgs.delve
        ];
        env.CGO_ENABLED = "0";
        env.GOTOOLCHAIN = "local";
      };
    };
}
```

```
nix develop --command go version
nix develop --command which gopls
```

`which go` outside the shell must not be this store path (or a Homebrew go you should not use).

Case 2: direnv

Save as `.envrc`:

```
# .envrc
use flake
```

```
direnv allow
go version
```

Commit `.envrc`. Gitignore `.direnv/`.

Case 3: Tests as you type

```
nix develop --command go test ./...
nix develop --command golangci-lint run
```

CI runs the same via `nix develop --command` or `buildGoModule doCheck`. Do not `go test` with a Homebrew Go.

A flake check:

```
# in outputs
{
  checks.x86_64-linux.lint = pkgs.runCommand "desk-lint" {
    nativeBuildInputs = [ pkgs.golangci-lint go ];
    src = ./.;
  } ''
    cd $src
    golangci-lint run
    touch $out
  '';
}
```

If lint needs the network, vendor fixtures; `checkPhase` is hermetic.

Case 4: CGO shell when you mean it

```
# cgo-shell.nix fragment
{
  packages = [ pkgs.go pkgs.pkg-config pkgs.sqlite ];
  env.CGO_ENABLED = "1";
}
```

Now `mattn/go-sqlite3` can compile. The **package** derivation needs the same `nativeBuildInputs` and `CGO_ENABLED`, or CI will not match.

Case 5: Private GOPRIVATE

```
# private.nix fragment
{
  packages = [ pkgs.go pkgs.git pkgs.openssh ];
  env.GOPRIVATE = "git.desk.internal";
  env.GOPROXY = "https://proxy.golang.org,direct";
}
```

SSH agent / `~/.netrc` is user state (sops / HM). The flake only names the host pattern.

The trap

The trap is **`go install golang.org/x/tools/gopls@latest` inside the shell**. That ignores the pin and writes to `GOPATH`. `pkgs.gopls` from 26.05.

The other trap is `GOTOOLCHAIN=auto` (Go default) downloading `go1.23.x` on first `go build`. `GOTOOLCHAIN=local`. Bump nixpkgs `go` (or overlay) if `go.mod` asks for newer.

The boring rule

- Same `go` in `mkShell` and `buildGoModule`.
- `gopls` / `golangci-lint` / `delve` from nixpkgs, not `@latest`.
- `GOTOOLCHAIN=local`. `CGO_ENABLED` matches the package.
- direnv `use flake`. No Home Manager `pkgs.go`.
- GOPRIVATE in the shell; tokens not in Nix.

Try this

1. `nix develop --command go env GOTOOLCHAIN GOVERSION CGO_ENABLED`.
2. `which go` inside vs outside direnv.
3. Add `golangci-lint run` as a `checks.` derivation.
4. Set `GOTOOLCHAIN=local` and a `go.mod` that asks for a newer go — read the error; bump nixpkgs `go` instead of letting Go download.

Cross-Compiling Go

Cross-Compiling Go

Go cross-compiles without a sysroot **when CGO is off**. The boring default is: **`CGO_ENABLED=0` and `GOOS/GOARCH` via `buildGoModule` env, or a native remote builder — not qemu-user for every PR**.

`pkgsCross` still matters if CGO is on (cgo + glibc for the target). For the desk API, turn CGO off first. This is not the C `pkgsCross` chapter: the compiler is still the **build** `go`; it merely emits another GOARCH.

Mental model

Need	Tool
linux/amd64 binary from a Mac	`GOOS=linux GOARCH=amd64`, CGO=0
linux/arm64 on x86_64, CGO=0	`GOARCH=arm64`
CGO + glibc for aarch64	`pkgsCross.aarch64-multiplatform.buildGoModule` or an aarch64 **builder**
NixOS test VM	Native `aarch64-linux` builder + KVM

```
buildGoModule {
  env.CGO_ENABLED = "0";
  env.GOOS = "linux";
  env.GOARCH = "arm64";
}
```

`file` the result. Flake `packages.x86_64-linux.desk-api-arm64` means **built on** x86_64, not that the ELF is amd64.

Worked examples

Case 1: linux/arm64 from x86_64, no CGO

Save as `go-arm.nix`:

```nix
# go-arm.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.buildGoModule {
```

```
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  env.CGO_ENABLED = "0";
  env.GOOS = "linux";
  env.GOARCH = "arm64";
}
```

```
nix-build go-arm.nix
file result/bin/desk-api
```

Output (shape):

```
result/bin/desk-api: ELF 64-bit LSB executable, ARM aarch64, statically linked, …
```

Do not run it on the x86_64 laptop. Copy to a Pi or use `qemu-user` once. Static + CGO=0 is why there is no target glibc in the closure.

Case 2: Flake packages per arch

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-api multi-arch";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
      srcGo = {
        pname = "desk-api";
        version = "1.0.0";
        src = ./.;
        vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
        env.CGO_ENABLED = "0";
        env.GOOS = "linux";
      };
    in
    {
      packages.x86_64-linux = {
        desk-api = pkgs.buildGoModule (srcGo // { env.GOARCH = "amd64"; });
        desk-api-arm64 = pkgs.buildGoModule (srcGo // { env.GOARCH = "arm64"; });
      };
```

```
    };
}
```

```
nix build .#desk-api-arm64
file result/bin/desk-api
```

CI: `nix build .#desk-api-arm64` on ubuntu-amd64. Seconds, not a QEMU VM.

Case 3: CGO — use pkgsCross or a builder

```
# go-cgo-arm.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.pkgsCross.aarch64-multiplatform.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = ./.;
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  env.CGO_ENABLED = "1";
}
```

Now you have a target gcc. Slow. Prefer `ssh-ng` to an ARM machine and native `buildGoModule`. Mixing `GOARCH=arm64` with host `gcc` is Case 5's failure mode.

Case 4: Windows / Darwin as GOOS

```
# go-darwin.nix fragment
{
  env.GOOS = "darwin";
  env.GOARCH = "arm64";
  env.CGO_ENABLED = "0";
}
```

```
nix-build
file result/bin/desk-api
```

```
Mach-O 64-bit executable arm64
```

Shipping a Mac binary from Linux CI is possible with CGO off. Notarization is not Nix's job. `GOOS=windows GOARCH=amd64` yields a `.exe` the same way.

Case 5: `file` and `go version -m`

```
file result/bin/desk-api
go version -m result/bin/desk-api | head
```

```
desk-api: go1.24.x
    path    desk/cmd/desk-api
    mod desk    v0.0.0
    build   GOARCH=arm64
    build   GOOS=linux
    build   CGO_ENABLED=0
```

If `linux/amd64` leaked, `env.GOARCH` did not apply (wrong attr, or CGO forced host).

The trap

The trap is **CGO on + `GOARCH=arm64` on an x86_64 host** without a cross gcc. The build uses the **host** gcc and produces a confused binary or a compile error. CGO off, or `pkgsCross`, or a native builder.

The other trap is qemu-user in every GitHub job. Minutes. Native `GOARCH` with CGO=0 is seconds.

The boring rule

- CGO off for services you cross.
- `env.GOOS` / `env.GOARCH` on `buildGoModule`. Probe `vendorHash`; never leave `fakeHash` on main.
- `file` and `go version -m` the output.
- CGO on → pkgsCross or remote native builder.
- Do not qemu the whole NixOS closure to get a Go binary.

Try this

1. Case 1; `file`; confirm aarch64.
2. Same drv with `GOARCH=amd64`; `file`; two different hashes.
3. Set `CGO_ENABLED=1` without a cross gcc; read the error.
4. `go version -m` on both binaries and confirm `GOARCH`.

Packaging Anti-Patterns

Packaging Anti-Patterns

Most "Nix is slow / undeterministic" reports are the same five habits. The boring default is: **fix these before adding Crane, uv2nix, or another overlay file**.

Mental model

Smell	Usual cause
CI laptop	`<nixpkgs>` or unlocked `nixpkgs#`
Every commit rebuilds gcc	`src = ./.` includes docs / `.git` / `target/`
Eval takes minutes	import-from-derivation
`go` on PATH CI	`nix-env` / Home Manager compiler
Infinite recursion	`final.pkg.overrideAttrs`
Password in `nix why-depends`	secret in a derivation

dream2nix / flake-parts are not the default. They paper over these if you let them. Delete the anti-pattern first.

Worked examples

Case 1: `<nixpkgs>` in production

Bad:

```
# configuration.nix - do not
{ pkgs ? import <nixpkgs> { }, ... }: {
  environment.systemPackages = [ pkgs.hello ];
}
```

Fix — pin nixos-26.05:

```
# flake.nix
{
  description = "Desk";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
```

```
    nixosConfigurations.desk = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [ ./configuration.nix ];
    };
  };
}
```

```
nix flake metadata
```

One nixpkgs node. Two nodes means a `follows` is missing.

Case 2: Fat `src`

Bad: `src = ./.;` at the monorepo root.

Fix:

```
# desk-api.nix
{ pkgs, lib }:

pkgs.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = lib.fileset.toSource {
    root = ../.;
    fileset = lib.fileset.unions [ ../apps/desk-api ../go.mod ../go.sum ];
  };
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build
echo x >> README.md
nix-build --dry-run
```

If gcc is in “will be built,” the filter failed.

Case 3: IFD

Bad:

```
# do not
let
  src = pkgs.fetchFromGitHub {
    owner = "desk";
    repo = "mod";
    rev = "abc123";
```

```
    hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  };
in
import "${src}/default.nix"
```

Eval **builds** `src` before it can parse `default.nix`. Flakes + IFD is a foot-gun.

Fix: copy `default.nix` into *this* repo, or add a flake input and `callPackage`.

Case 4: Compiler on the profile

Bad:

```
nix-env -iA nixpkgs.go
nix profile install nixpkgs#rustc
```

Fix: `devShells.default` with the same `go` / `rustPlatform` as `buildGoModule` / `buildRustPackage`.

```
# flake.nix fragment
{
  devShells.x86_64-linux.default = pkgs.mkShell {
    packages = [ pkgs.go pkgs.gopls ];
  };
}
```

`nix profile` for `jq`, not compilers. `which go` outside `nix develop` should fail (or be a distro go you must not use).

Case 5: Overlay recursion and secrets

Bad: `hello = final.hello.overrideAttrs (_: { })` → infinite recursion.

Fix: `prev.hello.overrideAttrs`.

Bad:

```
# do not
stdenv.mkDerivation {
  pname = "desk-secret";
  version = "1.0";
  src = ./.;
  password = "hunter2";
}
```

or `writeText "env" "TOKEN=…"`.

Fix: sops-nix path at **runtime**. Check:

```
nix path-info -r ./result | xargs -I{} grep -l hunter2 {} 2>/dev/null || echo 'token not in
```

`nix path-info -r` must not `grep` the token.

The trap

The trap is **papering over these with dream2nix / flake-parts / a mega-overlay**. The new tool then IFDs, too. Delete the anti-pattern first. The other trap is `fakeHash` left on `main` — every CI run is a FOD mismatch.

The boring rule

- Pin **nixos-26.05**. Commit `flake.lock`.
- Filter `src`. No IFD on the hot path.
- No `nix-env`. No compiler in `home.packages`.
- Override **this** package from `prev`.
- No secret bytes in `$out`.

Try this

1. `git grep -n '<nixpkgs>'` and `git grep 'src = \./\.'`.
2. `git grep nix-env`.
3. `nix flake metadata` — a single nixpkgs lock node.
4. `--dry-run` after a README-only edit on a filtered package.
5. `git grep -nE 'password = "|TOKEN=' -- '*.nix'`.

NixOS System

The NixOS Module System Architecture

The NixOS Module System Architecture

Editing `/etc` by hand and hoping two files agree is how machines drift. The boring default is: **the NixOS module system — typed options, merged `config`, and one evaluation that becomes a generation**.

You already installed a VM. This chapter is how that `configuration.nix` is not a special snowflake: it is one module among many, including the ones you will write for the desk.

Mental model

A module is a function:

```
{ config, lib, pkgs, ... }:
{
  options = { ... };  # what *may* be set (types, defaults, docs)
  config  = { ... };  # what *is* set (units, packages, files)
  imports = [ ... ];  # other modules, merged in
}
```

If you omit `options` and only set `config` keys (`services.openssh.enable = true;`), you are still a module — you are **consuming** options someone else declared (usually in nixpkgs).

The evaluator:

1. Loads every imported module (your files + nixpkgs NixOS modules).
2. Merges `options` (cannot define the same option twice).
3. Merges `config` with **priorities**.
4. Type-checks. Failures are eval errors, not "nginx failed at 3 a.m."

Helper	Role
`lib.mkEnableOption "…"`	Boolean `enable` with a description
`lib.mkOption { type, default, … }`	Any typed option
`lib.mkIf cond cfg`	Include `cfg` only when `cond` is true
`lib.mkDefault value`	Low priority (easy to override)
`lib.mkForce value`	High priority (wins over ordinary assignment)
`lib.mkMerge [ a b ]`	Merge two config attrsets

```
desk-module.nix (options + mkIf)
host.nix        (enable = true; port = 9090)
nixpkgs modules (openssh, users, …)
```

```
one config  →  one system derivation
```

Worked examples

Case 1: A small desk module

Save as `desk_module.nix`:

```nix
# desk_module.nix
{ lib, config, pkgs, ... }:

let
  cfg = config.services.deskOperations;
in
{
  options.services.deskOperations = {
    enable = lib.mkEnableOption "Desk operations banner";

    port = lib.mkOption {
      type = lib.types.port;
      default = 8080;
      description = "TCP port printed by the banner.";
    };

    banner = lib.mkOption {
      type = lib.types.str;
      default = "Desk Operational";
      description = "Text printed by desk-banner.";
    };
  };

  config = lib.mkIf cfg.enable {
    assertions = [
      {
        assertion = cfg.port != 22;
        message = "deskOperations.port must not steal sshd's 22";
      }
    ];
    environment.systemPackages = [
      (pkgs.writeShellScriptBin "desk-banner" ''
        echo "${cfg.banner} on port ${toString cfg.port}"
      '')
    ];
  };
```

```
}
```

`cfg.enable` is false by default. Nothing is installed until a host turns it on. That is the whole point of `mkIf`.

Case 2: Consume it from a host

Save as `host.nix`:

```
# host.nix
{
  imports = [ ./desk_module.nix ];

  services.deskOperations = {
    enable = true;
    port = 9090;
    banner = "Window A online";
  };
}
```

On a NixOS flake (or a lab `nixosSystem` eval):

```
nix eval .#nixosConfigurations.desk-vm.config.services.deskOperations.port
```

Output:

```
9090
```

The option is an integer (`lib.types.port`), not a string. Passing `"9090"` is an eval error.

Case 3: Priorities — `mkDefault` versus `mkForce`

Save as `priorities.nix`:

```
# priorities.nix
{ lib, ... }:

{
  networking.hostName = lib.mkDefault "generic-desk";
  services.openssh.enable = lib.mkForce true;
}
```

Save as `host-override.nix`:

```
# host-override.nix
{
```

```
  imports = [ ./priorities.nix ];
  networking.hostName = "desk-vm";
}
```

Ordinary assignment (`"desk-vm"`) beats `mkDefault`. It does **not** beat `mkForce`. If a later file sets `services.openssh.enable = false;`, evaluation **fails** (conflict) unless that file also uses `mkForce`.

Use `mkDefault` in shared profiles (corp baseline). Use `mkForce` rarely: "this host *must* have sshd," not as a habit.

`//` on two config attrsets **replaces** nested keys. `lib.mkMerge [ a b ]` **merges** them the module way (lists concatenate, priorities apply). Shared modules that `config = lib.mkMerge [ … ]` compose; `config = parent // { … }` is how you accidentally drop `wantedBy`.

`lib.mkBefore` / `lib.mkAfter` order list items (`environment.etc` lines, `systemd` drop-ins). Reach for them when order is the bug, not when you meant `mkForce`.

Case 4: Types catch mistakes at eval

Save as `bad-port.nix`:

```
# bad-port.nix
{
  imports = [ ./desk_module.nix ];
  services.deskOperations = {
    enable = true;
    port = "ninety";
  };
}
```

```
nixos-rebuild dry-build --flake .#desk-vm
```

Output (shape):

```
error: A definition for option `services.deskOperations.port' is not of type `16 bit unsigned i
```

No unit started. No half-applied `/etc`. That is the module system paying rent.

Case 5: Inspect options without guessing

```
nixos-option services.openssh.enable
nixos-option services.deskOperations.port
```

On a flake host:

```
nix eval .#nixosConfigurations.desk-vm.options.services.deskOperations.port.description
```

`man configuration.nix` (or `nixos-help`) is the same data as HTML. Prefer `nixos-option` / `nix eval` when you want the **live** merged value, not the upstream default.

The trap

The trap is `type = lib.types.anything` (or no type) so "it just merges." Then `port = "8080";` ships, the script interpolates a string, and the unit fails at boot. Types are the boring check.

The other trap is defining `options.services.openssh.enable` yourself. That option already exists. You **set** it; you do not redeclare it. Redeclare → "option already defined."

A third trap: putting implementation in `options` (an option whose default starts a service). Defaults should be data. Side effects belong in `config = mkIf cfg.enable { … }`.

The boring rule

- One concern per module. Import it from the host. Do not paste the same 80 lines into three `configuration.nix` files.
- `mkIf cfg.enable` around every implementation.
- `mkDefault` in shared profiles. `mkForce` only when a conflict is the point.
- Strict types (`bool`, `port`, `str`, `listOf str`, `submodule`). Never `anything` for desk options.
- `assertions` for invariants (port 22). They fail at eval, not at 3 a.m.
- `mkMerge`, not `//`, when composing config. `mkBefore`/`mkAfter` for list order.
- Inspect with `nixos-option` / `nix eval`, not by reading `/etc` after the fact.

Try this

1. Add `workers = lib.mkOption { type = lib.types.ints.positive; default = 4; };` and print it from `desk-banner`.
2. Set `port = 70000` and read the type error.
3. Put `networking.hostName = lib.mkDefault "a";` in one file and `networking.hostName = "b";` in another. Eval. Then change the second to `mkDefault "b"` and see the conflict.
4. `nixos-option services.openssh.settings.PasswordAuthentication` (or eval the flake option) and write down the default. Confirm it matches what you intend to set in the SSH chapter.

Configuring Your First NixOS Machine

Configuring Your First NixOS Machine

The installer left you with `/etc/nixos/configuration.nix` and a generation 1. The boring default from here is: **move that config into a flake in git, keep `hardware-configuration.nix` generated, and treat `nixos-rebuild switch --flake` as the only mutation path**.

The Installing NixOS chapter got a VM to boot. This chapter is how that VM becomes a **host you can clone**.

Mental model

File	Who writes it	How often it changes
`hardware-configuration.nix`	`nixos-generate-config`	When disks or initrd modules change
`configuration.nix` / host module	You	Every service, user, package
`flake.nix` + `flake.lock`	You	Pin nixpkgs; add the host output
`/run/current-system`	`nixos-rebuild`	Every successful switch

Rebuild verbs:

Command	Activates now?	Bootloader default?	Operational use case
`switch`	Yes	Yes	Confirmed changes ready for daily use
`test`	Yes	**No**	Firewall, network, or SSH tweaks (reboot recovers state)
`boot`	No	Yes	Kernel upgrades, sysctls, or maintenance during active hours
`dry-activate`	No	No	Inspect systemd unit restart diffs before applying
`build`	No	No	Syntax and derivation build check (writes `./result`)

```
git flake  --nixos-rebuild switch-->  new generation
                                        /nix/store/…-nixos-system-…
                                        bootloader entry
                                        /run/current-system
```

`system.stateVersion` stays at the **birth** release. It is not "which channel I follow."

Worked examples

Case 1: A readable host module

Save as `configuration.nix` (or `hosts/desk-vm.nix` in a flake):

```nix
# configuration.nix
{ config, pkgs, ... }:

{
  imports = [ ./hardware-configuration.nix ];

  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;

  networking.hostName = "desk-workstation";
  time.timeZone = "UTC";
  i18n.defaultLocale = "en_US.UTF-8";

  environment.systemPackages = with pkgs; [
    git
    ripgrep
    htop
    curl
  ];

  services.openssh.enable = true;

  system.stateVersion = "26.05";
}
```

Leave `hardware-configuration.nix` as generated. Do not pretty-print UUIDs into `/dev/vda2`.

Case 2: Flake output for the same host

Save as `flake.nix` next to those files:

```
# flake.nix
{
  description = "Desk workstation";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.desk-workstation = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [ ./configuration.nix ];
    };
  };
}
```

The output **name** should match `networking.hostName` (or be a short alias you document). Rebuild:

```
sudo nixos-rebuild switch --flake .#desk-workstation
```

Output:

```
building the system configuration...
updating systemd-boot...
activating the configuration...
setting up /etc...
```

`--flake .#desk-workstation` reads `flake.lock`. Two machines with the same lock build the same closure (hardware module aside).

Case 3: `test` before `switch`

Add `cowsay` to `environment.systemPackages`, then:

```
sudo nixos-rebuild test --flake .#desk-workstation
cowsay desk
```

Output:

```
 ----
< desk >
 ----
```

Reboot. `cowsay` is **gone**: `test` did not update the bootloader default. That is the point — try a service, reboot if it bricks the session, land on the last `switch`.

When you like it:

```
sudo nixos-rebuild switch --flake .#desk-workstation
```

Case 4: Dry-activate, generations, and bootloader recovery

```
sudo nixos-rebuild dry-activate --flake .#desk-workstation
nixos-rebuild list-generations
```

Output (shape):

```
would restart the following units: …
would start the following units: …

Generation  Build-date           NixOS version          Configuration
  1         2026-09-01 10:00:00  26.05.9440.6aefcda
  2         2026-09-09 12:00:00  26.05.9440.6aefcda     (current)
```

`dry-activate` is how you see that an nginx change restarts nginx and does **not** restart sshd. Read it before `switch` on a machine people are using.

Rollback in a running shell:

```
sudo nixos-rebuild switch --rollback
```

If a bad change breaks networking or panics early boot before you get a shell, do not hunt for a live USB: **reboot the physical or virtual host and select Generation 1 in the systemd-boot (or GRUB) menu**. NixOS stores previous closures until you run garbage collection.

Case 5: Scaling to a multi-node homelab (`hosts/` and `modules/`)

When you add a second machine (e.g. `homelab-docker`), do not clone a giant monolithic `configuration.nix`. Split your repository into host identities and reusable role modules:

```
.
  flake.nix
  flake.lock
  hosts/
      desk-workstation/
          default.nix
          hardware-configuration.nix
      homelab-docker/
          default.nix
          hardware-configuration.nix
  modules/
      common.nix
      docker-host.nix
```

Save baseline settings in `modules/common.nix`:

```
# modules/common.nix
{ pkgs, ... }:

{
  nix.settings.experimental-features = [ "nix-command" "flakes" ];

  time.timeZone = "UTC";
  i18n.defaultLocale = "en_US.UTF-8";

  environment.systemPackages = with pkgs; [
    git
    ripgrep
    htop
    curl
    tmux
  ];

  services.openssh = {
    enable = true;
    settings.PasswordAuthentication = false;
    settings.KbdInteractiveAuthentication = false;
  };
}
```

Save reusable server roles in `modules/docker-host.nix`:

```
# modules/docker-host.nix
{ pkgs, ... }:

{
  virtualisation.docker = {
    enable = true;
    autoPrune = {
      enable = true;
      dates = "weekly";
    };
  };

  environment.systemPackages = with pkgs; [
    docker-compose
  ];
}
```

Compose both machines in `flake.nix`:

```
# flake.nix
{
  description = "Homelab infrastructure";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations = {
      desk-workstation = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ./hosts/desk-workstation
          ./modules/common.nix
        ];
      };

      homelab-docker = nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          ./hosts/homelab-docker
          ./modules/common.nix
          ./modules/docker-host.nix
        ];
      };
    };
  };
}
```

Rebuilding any specific node in the fleet is just pointing to its target attr:

```
sudo nixos-rebuild switch --flake .#homelab-docker
```

Case 6: Declarative package hygiene

In conventional Linux distributions, machines accumulate hundreds of forgotten packages via `apt install` or `dnf install`. On NixOS, the equivalent mistake is turning `environment.systemPackages` into a dumping ground.

Maintain strict three-tier package hygiene:

1. **Host-wide baseline (`modules/common.nix`)**: Only tools required on every node for triage and administration (`git`, `curl`, `htop`, `tmux`, `ripgrep`).
2. **Role packages (`modules/docker-host.nix`, `modules/db.nix`)**: Packages tied directly to that role's operation (e.g. `docker-compose` or `postgresql_17`).
3. **Ad-hoc exploratory tools (no system rebuild)**: For one-off diagnostics, benchmarks, or utilities (`iperf3`, `nmap`, `fastfetch`), do **not** edit your configuration. Run them ephemerally:

```
nix run nixpkgs#iperf3 -- -s
nix shell nixpkgs#nmap -c nmap 192.168.1.0/24
```

These download and run directly from the Nix store without polluting the host closure or creating a new system generation.

The trap

The trap is **changing `system.stateVersion` when you bump nixpkgs to 26.11**. Databases, `stateVersion`-gated defaults, and Home Manager state dirs then migrate in surprising ways. Follow the release notes. Keep the birth value until a note tells you to bump a **specific** option.

Another trap is running `nixos-rebuild switch` over SSH when modifying firewall rules, network interfaces, or SSH daemon options. If a rule syntax or port binding fails, you are severed from the machine. Always run `nixos-rebuild test` for remote connectivity changes: a hard reboot or hypervisor reset will immediately restore the previous generation.

A third trap: editing `/etc/nixos` on the live machine and never pushing git. The next `nixos-anywhere` or the next laptop rebuilds last week's host. The flake in git is the machine. `/etc/nixos` is either a checkout or a leftover from the installer — pick one.

A fourth trap: `nixos-rebuild switch` **without** `--flake` after you moved to a flake. You just built the old `/etc/nixos/configuration.nix` against channels. `nixos-version` and `nix flake metadata` should agree.

`boot.loader.systemd-boot.configurationLimit = 8;` so `/boot` does not fill. `nix.gc.automatic` with `--delete-older-than 14d` so the store does not. Neither is a substitute for `git` being the machine.

The boring rule

- Flake in git. `--flake .#host`. Lockfile committed.
- `hardware-configuration.nix` generated; host module written by you.
- `test` for network/firewall/SSH changes; `boot` for kernel updates during working hours; `switch` when confirmed; `dry-activate` before production restarts.
- Organize fleets with `hosts/<name>/default.nix` and shared `modules/`. No framework needed.
- Keep `systemPackages` lean. Use `nix run` or `nix shell` for ad-hoc exploration.
- ESP `configurationLimit` + 14-day GC. Still `--flake .#host`.
- `stateVersion` is a birth certificate.

Try this

1. `nixos-version` and `nix flake metadata` — confirm the nixpkgs revision in the lock matches what you think is running.

2. `nixos-rebuild list-generations` to inspect your current generation history and timestamps.
3. `sudo nixos-rebuild test` with a hostname change, `hostname`, reboot, `hostname` again. Then `switch` if you want it permanent.
4. Run `nix run nixpkgs#fastfetch` directly without adding it to `environment.systemPackages`. Verify that `which fastfetch` reports nothing after the command exits.
5. `sudo nixos-rebuild dry-activate` after adding `services.nginx.enable = true;` (lab VM). Read which units would start. Disable nginx if you do not need it.

Service Management with systemd

Service Management with systemd

Hand-written unit files in `/etc/systemd/system/` plus `daemon-reload` is how two hosts disagree. The boring default is: **`systemd.services.<name>` as a Nix attrset, store-path binaries, and a high-level `services.<app>.enable` module when nixpkgs already has one**.

Mental model

NixOS renders unit files into the system closure. `nixos-rebuild switch` diffs old vs new units and restarts what changed. You do not `systemctl enable` by hand; `wantedBy = [ "multi-user.target" ];` is enablement.

Two layers:

Layer	When
`services.nginx.enable = true;`	nixpkgs already knows nginx. Use it.
`systemd.services.desk-worker = { … };`	The desk binary has no module yet.

Custom units need **absolute** `ExecStart`. The service `PATH` is not your login `PATH`. `${pkgs.coreutils}/bin/echo`, not `echo`.

Hardening that costs one line:

Setting	Effect
`DynamicUser = true;`	Ephemeral uid; no `useradd`
`ProtectSystem = "strict";`	Root filesystem read-only
`ProtectHome = true;`	`/home` inaccessible
`PrivateTmp = true;`	Private `/tmp`
`NoNewPrivileges = true;`	No privilege escalation

```
systemd.services.desk-api
   wantedBy  →  starts at boot
   after     →  ordering
   ExecStart →  store path
   Restart   →  on-failure
```

Worked examples

Case 1: A desk worker unit

Save as `desk_service_unit.nix`:

```
# desk_service_unit.nix
{ pkgs, ... }:

{
  systemd.services.desk-worker = {
    description = "Desk operations worker";
    wantedBy = [ "multi-user.target" ];
    after = [ "network-online.target" ];
    wants = [ "network-online.target" ];
    # network.target is "stack is up", not "have a default route".
    # Bind-to-0.0.0.0 workers can use network.target. Outbound HTTP needs online.

    environment = {
      DESK_ENVIRONMENT = "production";
      PORT = "8080";
    };

    serviceConfig = {
      Type = "simple";
      ExecStart = "${pkgs.writeShellScript "desk-worker" ''
        ${pkgs.coreutils}/bin/echo "Desk worker on port $PORT"
        while true; do ${pkgs.coreutils}/bin/sleep 3600; done
      ''}";
      Restart = "on-failure";
      RestartSec = "5s";
      DynamicUser = true;
      ProtectSystem = "strict";
      ProtectHome = true;
      NoNewPrivileges = true;
      PrivateTmp = true;
      MemoryMax = "256M";
    };
  };
}
```

```
sudo nixos-rebuild switch
systemctl status desk-worker
journalctl -u desk-worker -n 20 --no-pager
```

Output (shape):

```
desk-worker.service - Desk operations worker
    Loaded: loaded (/etc/systemd/system/desk-worker.service; enabled)
    Active: active (running)
```

`/etc/systemd/system/desk-worker.service` is a symlink into `/nix/store`. Editing it by hand is undone on the next switch.

Case 2: Prefer a nixpkgs module when it exists

Save as `nginx.nix`:

```
# nginx.nix
{ pkgs, ... }:

{
  services.nginx = {
    enable = true;
    virtualHosts."desk.internal" = {
      root = pkgs.writeTextDir "index.html" "<h1>Desk</h1>";
    };
  };
}
```

This opens the right units, user, and (optionally) firewall helpers. Do not copy a 40-line `systemd.services.nginx` from a wiki.

```
systemctl is-active nginx
curl -sS http://127.0.0.1/ | head
```

Case 3: Timers instead of cron

Save as `desk_timer.nix`:

```
# desk_timer.nix
{ pkgs, ... }:

{
  systemd.services.desk-cleanup = {
    description = "Clean stale desk tickets";
    serviceConfig = {
      Type = "oneshot";
      ExecStart = "${pkgs.coreutils}/bin/echo desk-cleanup-ok";
    };
  };

  systemd.timers.desk-cleanup = {
```

```
    wantedBy = [ "timers.target" ];
    timerConfig = {
      OnCalendar = "daily";
      Persistent = true;
    };
  };
}
```

```
systemctl list-timers --all | grep desk
sudo systemctl start desk-cleanup.service
journalctl -u desk-cleanup -n 5 --no-pager
```

`Persistent = true` catches up after the VM slept through midnight.

Case 4: Secrets as files, not unit text

```
# desk-envfile.nix
{
  systemd.services.desk-worker.serviceConfig.EnvironmentFile =
    "/run/secrets/desk-worker.env";
}
```

`environment.PORT = "8080";` is fine for non-secrets. A database password in `environment.PASSWORD = "…"` lands in the world-readable store. `EnvironmentFile` points at a sops-decrypted tmpfs path (secrets part). The unit file then contains a **path**, not the secret.

`LoadCredential = "db:${config.sops.secrets.db.path}";` is the systemd-native sibling: the credential appears under `$CREDENTIALS_DIRECTORY/db` with mode `0400` for the service uid. Prefer it over leaking `Environment=` when the app can read a file. `Type = "notify"` only if the binary calls `sd_notify`; a shell `while sleep` is `simple`. A oneshot timer job is `Type = "oneshot"` (Case 3) — `Restart = "always"` on a oneshot is a fork bomb.

Case 5: See what switch will restart

```
sudo nixos-rebuild dry-activate --flake .#desk-workstation
```

Output (shape):

```
would restart the following units: desk-worker.service
would not restart: sshd.service
```

If `dry-activate` says it will restart sshd because you changed a comment in a shared module, split the module. Restarts are part of the change.

The trap

The trap is **`ExecStart = "desk-api"`** or `ExecStart = "/usr/bin/curl …"`**. The unit's PATH is minimal. `desk-api: command not found` at boot, while your interactive shell still has it.

Always `${pkgs.desk-api}/bin/desk-api` or a `writeShellScript` that uses store paths inside.

The other trap is `systemctl edit desk-worker` drop-ins. They live outside the flake. The next switch may fight them, or they survive and surprise the next operator. If you needed the drop-in, it belonged in `systemd.services.desk-worker.serviceConfig`.

The boring rule

- High-level `services.<app>` when nixpkgs has it. Raw `systemd.services` for desk-specific units.
- `ExecStart` is a store path. No bare names, no `/usr/bin`.
- `DynamicUser` + `ProtectSystem = "strict"` on network daemons that do not need a named uid.
- Timers, not crontab.
- Secrets via `EnvironmentFile` / `LoadCredential` / sops paths, never `environment.SECRET`.
- `network-online.target` only when you need a route. `Type=notify` only when the app notifies.

Try this

1. Add `serviceConfig.MemoryMax = "64M";` to the worker, switch, `systemctl show desk-worker -p MemoryMax`.
2. Change `ExecStart` to `echo hello` (no store path), switch, `journalctl -u desk-worker -n 20`. Restore the store path.
3. `systemctl cat desk-worker` and confirm the unit lives under `/nix/store`.
4. Enable `services.timesyncd` (or `services.chrony`) via the **module**, not a custom unit. `systemctl status systemd-timesyncd`.

Networking, Firewall, and Security

Networking, Firewall, and Security

Ad-hoc `iptables -A` and a NetworkManager GUI click are gone after a rebuild — or worse, they **survive** in unmanaged `/etc` and fight the next activation. The boring default is: **firewall on, ports listed in Nix, NetworkManager on laptops, systemd-networkd on servers, secrets as files**.

Mental model

NixOS enables a firewall by default (`networking.firewall.enable` is true). Incoming is denied unless you or a module open it. `services.openssh.enable = true` opens 22 unless you tell it not to (`openFirewall = false` on modules that offer that flag).

Machine	Network backend
Desk laptop, Wi-Fi	`networking.networkmanager.enable = true;`
VM, cloud, rack server	`networking.useNetworkd = true;` (systemd-networkd)

Do not enable both and hope. Pick one.

26.05 uses **nftables** for `networking.firewall` unless you set `networking.nftables.enable = false` (legacy iptables). `nft list ruleset` is the boring inspect command. `networking.firewall.extraInputRu` is an nft snippet when a typed option is missing — still better than a page of `extraCommands`.

```
internet


nftables/iptables  ←  networking.firewall.allowedTCPPorts


sshd / nginx / desk-api
```

WireGuard, Tailscale, and similar are **modules** (`networking.wireguard`, `services.tailscale`). They are not extra packages you `systemctl start` by hand.

Worked examples

Case 1: Hostname, DNS, firewall allow-list

Save as `networking_config.nix`:

```
# networking_config.nix
{
  networking = {
    hostName = "desk-gateway";
    domain = "internal.corp";

    firewall = {
      enable = true;
      allowPing = true; # ICMP; set false on a bastion if you mean it
      allowedTCPPorts = [ 22 8080 ];
      allowedTCPPortRanges = [
        { from = 9000; to = 9010; }
      ];
    };

    nameservers = [ "1.1.1.1" "9.9.9.9" ];
  };
}
```

```
sudo nixos-rebuild switch
sudo nft list ruleset | head
# or, if still on iptables backend:
sudo iptables -L INPUT -n | head
```

Port 8080 is open because you listed it. Port 3000 is not. A node app that binds `:3000` is reachable on localhost only until you add 3000.

Case 2: systemd-networkd static address (server)

Save as `networkd.nix`:

```
# networkd.nix
{
  networking.useDHCP = false;
  networking.useNetworkd = true;
  networking.networkmanager.enable = false;

  systemd.network.networks."10-lan" = {
    matchConfig.Name = "en*";
    networkConfig = {
```

```
      Address = "10.20.0.10/24";
      Gateway = "10.20.0.1";
      DNS = "10.20.0.1";
    };
  };
}
```

Match on `en*` (or a `by-id` name) in the lab. On metal, prefer `matchConfig.MACAddress` or `Name = "enp1s0"` **after** you have seen `networkctl status`.

```
networkctl status
ip -br addr
```

Case 3: NetworkManager on the workstation

Save as `nm.nix`:

```
# nm.nix
{
  networking.networkmanager.enable = true;
  networking.useNetworkd = false;
  networking.firewall.enable = true;
}
```

Users who should click Wi-Fi:

```
users.users.deskadmin.extraGroups = [ "networkmanager" ];
```

Do not also set `networking.interfaces.eth0.ipv4.addresses` unless you know you want NixOS's scripted networking **instead**. One backend.

Case 4: WireGuard with a key file

Save as `wireguard.nix`:

```
# wireguard.nix
{
  networking.firewall.allowedUDPPorts = [ 51820 ];

  networking.wireguard.interfaces.wg0 = {
    ips = [ "10.100.0.2/24" ];
    listenPort = 51820;
    privateKeyFile = "/run/secrets/wg-desk.key";

    peers = [
```

```
      {
        publicKey = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        allowedIPs = [ "10.100.0.0/24" ];
        endpoint = "vpn.desk.internal:51820";
        persistentKeepalive = 25;
      }
    ];
  };
}
```

`privateKeyFile` is a **path**. The key bytes are not in the flake. Generate with `wg genkey` on the machine (or decrypt with sops) and persist that file (impermanence: `/persist`).

```
sudo systemctl status wg-quick-wg0
# or wg0 depending on the module's unit name
ip -br addr show wg0
```

Case 5: Trust a VPN interface without opening the world

```
# trusted.nix
{
  networking.firewall = {
    enable = true;
    trustedInterfaces = [ "lo" "wg0" ];
    allowedTCPPorts = [ 22 ];
  };
}
```

Traffic **from** `wg0` is trusted. The public NIC still only has 22. That is how the desk API can listen on `10.100.0.2:8080` without listing 8080 on the internet.

Per-interface holes (no trust of the whole overlay):

```
{
  networking.firewall.interfaces.wg0.allowedTCPPorts = [ 8080 5432 ];
}
```

Postgres on `wg0` only. `checkReversePath = "loose";` is for asymmetric routing (some VPNs); `"strict"` (default on many setups) drops martian sources. Do not flip it to `"loose"` to "make WireGuard work" until `nft` / `journalctl -k` says rpfilter is the drop.

The trap

The trap is **`networking.firewall.enable = false;`** "to debug the app," then committing the host. The next deploy is an open machine.

Open `allowedTCPPorts = [ 3000 ];` instead. Leave the firewall on.

The other trap is putting a WireGuard **private** key in the attrset (`privateKey = "…"`). World-readable store. Same as a password in `configuration.nix`.

A third trap: `networking.firewall.extraCommands` with a page of `iptables`. Those rules are not typed, not easily reviewed, and break when the backend is nftables. Prefer `allowedTCPPorts`, `interfaces.<name>.allowedTCPPorts`, and `trustedInterfaces`.

A fourth trap: deploying firewall rules or SSH port changes over SSH using `nixos-rebuild switch` directly. If you typo a port or misconfigure an interface, your connection drops and the unbootable network state is saved as default. Use `nixos-rebuild test --flake .#host` first; if you lose access, a hypervisor or out-of-band reboot automatically reverts the machine.

The boring rule

- Firewall stays on. List ports. Do not disable it to debug.
- NetworkManager **or** networkd, not both.
- VPN keys as `privateKeyFile` (sops / persist), never as Nix strings.
- Test firewall and SSH updates with `nixos-rebuild test` before committing with `switch`.
- Let service modules open their own ports, or set `openFirewall` deliberately.
- `trustedInterfaces` for overlay NICs; or `interfaces.<name>.allowedTCPPorts` if you do not trust the whole mesh.
- nftables backend. `nft list ruleset`. `checkReversePath` stays strict until proven.

Try this

1. Add `8080` to `allowedTCPPorts`, switch, `nc -l 8080` in one shell and `nc $HOST 8080` from another VM. Remove 8080 and confirm the second connect hangs/refuses.
2. `sudo nft list ruleset | grep -i 22` (or iptables) after `services.openssh.enable = true` with default firewall — 22 should be there.
3. Set `allowPing = false` on a lab VM, `ping` it from the host, then set it true again.
4. Generate a WireGuard keypair with `wg genkey | tee /tmp/k | wg pubkey` and **do not** paste the private half into a `.nix` file. Put it in a file and reference the path.

Users, Groups, and SSH Configuration

Users, Groups, and SSH Configuration

`useradd` on a server is an account git does not know about. The boring default is: **`users.mutableUsers = false`, SSH keys in the flake, password hashes from `mkpasswd` or a file, and `sshd` with password auth off**.

Mental model

With `mutableUsers = false`, every `nixos-rebuild switch` **rewrites** `/etc/passwd` and `/etc/shadow` from the config. An account that left the flake left the machine. `passwd` as a user may appear to work until the next switch (or be blocked). That is the feature.

Field	Use
`isNormalUser = true;`	Human: home, shell, extraGroups
`isSystemUser = true;`	Daemon uid you named (rare if `DynamicUser` suffices)
`extraGroups = [ "wheel" ];`	sudo (if sudo is enabled)
`openssh.authorizedKeys.keys`	Login without a password
`hashedPassword`	Hash **in** the store (still not plaintext)
`hashedPasswordFile`	Hash **outside** the store (better)
`initialPassword`	Lab only. Plaintext in the store. Never production.

```
flake  →  users.users.deskadmin
                uid/gid
                ~/.ssh/authorized_keys (generated)
                hashedPasswordFile
```

`services.openssh` is the daemon. User keys do not start sshd. Enable the service, then disable password authentication.

Worked examples

Case 1: Immutable users and hardened sshd

Save as `users_ssh.nix`:

```
# users_ssh.nix
{
  users.mutableUsers = false;

  users.users.deskadmin = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    openssh.authorizedKeys.keys = [
      "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskAdminLaptop deskadmin@laptop"
    ];
  };

  users.users.deployer = {
    isNormalUser = true;
    description = "CI deploy key";
    extraGroups = [ "wheel" ];
    openssh.authorizedKeys.keys = [
      "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleDeployerKeyCorp"
    ];
  };

  services.openssh = {
    enable = true;
    settings = {
      PasswordAuthentication = false;
      KbdInteractiveAuthentication = false;
      PermitRootLogin = "no";
      X11Forwarding = false;
    };
  };

  security.sudo.wheelNeedsPassword = true;
}
```

`security.sudo.wheelNeedsPassword = false` is convenient on a lab VM and a foot-gun on a laptop someone might walk away from. Default to needing a password (or a key + `sudo` timestamp) unless this is a CI image.

```
sudo nixos-rebuild switch
getent passwd deskadmin
sudo grep -E 'PasswordAuthentication|PermitRootLogin' /etc/ssh/sshd_config
```

Case 2: Hash a console password

On a trusted machine:

```
mkpasswd -m sha-512
```

Output (shape):

`$6$rounds=656000$SALT$HASH…`

```
# password_hash.nix
{
  users.users.operator = {
    isNormalUser = true;
    hashedPassword = "$6$rounds=656000$SALT$HASH";
  };
}
```

The hash is in the store. That is weaker than `hashedPasswordFile` but **not** the same as `initialPassword = "hunter2"`. Anyone with store access can **offline-crack** the hash; they do not see the password in `grep`. Prefer a file:

```
{
  users.users.operator.hashedPasswordFile = "/persist/secrets/operator.hash";
}
```

Generate the file once, mode `0400`, persist it (impermanence). sops-nix can decrypt it at boot.

Case 3: Root is not a daily login

```
# root.nix
{
  users.users.root.openssh.authorizedKeys.keys = [ ];
  services.openssh.settings.PermitRootLogin = "no";
}
```

Break-glass in a lab VM can use a root password hash **on the console**, not over SSH. Production: serial console + the previous generation, not `PermitRootLogin = "yes"`.

Case 4: Groups for devices, not for folklore

```
# groups.nix
{
  users.users.deskadmin.extraGroups = [
    "wheel"
    "networkmanager" # if NM is enabled
    "video"
    "dialout"        # USB serial, lab only
  ];
```

```
}
```

Each group is a capability. `extraGroups = [ "wheel" "docker" "libvirtd" "adbusers" "…"]` on every human is how laptops become root. Add groups when a device or module requires them.

`users.groups.desk.members = [ "deskadmin" "deployer" ];` for a shared directory.

Case 5: Prove password SSH is dead

From another machine:

```
ssh -o PreferredAuthentications=password -o PubkeyAuthentication=no deskadmin@desk-workstati
```

Output:

```
deskadmin@desk-workstation: Permission denied (publickey).
```

Then:

```
ssh -i ~/.ssh/id_ed25519 deskadmin@desk-workstation true
```

Output: silence, exit 0.

If password still works, `PasswordAuthentication` did not land (wrong `settings.` attr on this release) or you are not hitting this sshd.

```
# ssh-allow.nix fragment
{
  services.openssh.settings.AllowUsers = [ "deskadmin" "deployer" ];
  services.openssh.ports = [ 22 ];
}
```

`AllowUsers` is a second gate after keys. User systemd (`loginctl enable-linger deskadmin`) is **not** implied by `isNormalUser` — set `users.users.deskadmin.linger = true;` only if that user must run lingering `systemd --user` units after logout (CI agents, not laptops).

The trap

The trap is `initialPassword = "changeme"` left in the flake after install. The string is in `/nix/store` forever (and in git history). Even after you "change" it with `passwd`, `mutableUsers = false` may reset or ignore that, and the store still has `changeme`.

Use `hashedPassword` / `hashedPasswordFile`. Delete `initialPassword` from the first commit that leaves the lab.

The other trap is `users.mutableUsers = true` on a server "so the new hire can `passwd`." They never appear in git. Six months later nobody can say who has a shell. `false` plus a PR that adds their key.

The boring rule

- `mutableUsers = false` on anything you would call production (and on the lab VM, so you practise).
- SSH keys in the flake. Password SSH off. Root SSH off.
- Hashes from `mkpasswd -m sha-512`. Prefer `hashedPasswordFile`.
- Never `initialPassword` outside a throwaway installer snippet.
- `wheel` is sudo. Do not sprinkle extra groups "just in case."
- `AllowUsers` on sshd. `linger` only for user units that must outlive the session.

Try this

1. Run `mkpasswd -m sha-512`, paste the hash on a lab user, switch, log in on the VM console.
2. Add a second authorized key, switch, `ssh-add -l` / connect with that key. Remove the key, switch, confirm that key is refused.
3. `sudo useradd leftover` on a `mutableUsers = false` host, then `sudo nixos-rebuild switch`, then `getent passwd leftover`. The account should be gone.
4. `nix-store --query --references /run/current-system | xargs grep -l initialPassword` — should print nothing. If it prints, you still have plaintext in the closure.

Hardware Configuration and Drivers

Hardware Configuration and Drivers

The host module is identity (hostname, users, services). Hardware is silicon (disks, CPU, GPU, firmware). The boring default is: **generated `hardware-configuration.nix`, UUIDs not `/dev/sda`, CPU microcode on, and `nixos-hardware` when a profile exists for this chassis**.

Mental model

```
flake
  modules = [
    ./configuration.nix          # you
    ./hardware-configuration.nix # generated
    nixos-hardware.…             # community, optional
  ]
```

`nixos-generate-config --root /mnt` (installer) or `--show-hardware-config` (running system) writes filesystem UUIDs and `boot.initrd.availableKernelModules` (nvme, virtio, ahci). If initrd lacks the disk driver, the VM boots to an emergency shell.

Piece	Typical source
File systems, swap, initrd modules	`hardware-configuration.nix`
CPU microcode	`hardware.cpu.intel/amd.updateMicrocode`
Laptop quirks (audio, fingerprint)	`nixos-hardware`
GPU	`hardware.graphics` + vendor module if needed
Firmware blobs	`hardware.enableRedistributableFirmware = true;`

Do not merge hardware into `configuration.nix` "to have one file." The next disk clone will fight you.

On 26.05, GPU is `hardware.graphics.enable = true;` (the old `hardware.opengl` name is an alias). Intel/AMD usually need no extra blob beyond `enableRedistributableFirmware`. NVIDIA is a **policy** (`hardware.nvidia.open`, `prime`) — do not copy a 22.11 `videoDrivers = [ "nvidia" ];` gist. Disko already owns partition UUIDs; if you use Disko, do not also hand-write `fileSystems."/"` in hardware-configuration for the same mount.

Worked examples

Case 1: Read the generated file; do not invent it

Save as `hardware-configuration.nix` (illustrative — **your** UUIDs come from `blkid`):

```
# hardware-configuration.nix
{ config, lib, pkgs, modulesPath, ... }:

{
  imports = [ (modulesPath + "/installer/scan/not-detected.nix") ];

  boot.initrd.availableKernelModules = [ "nvme" "xhci_pci" "ahci" "usbhid" "sd_mod" ];
  boot.kernelModules = [ "kvm-amd" ];

  fileSystems."/" = {
    device = "/dev/disk/by-uuid/a1b2c3d4-e5f6-7890-abcd-1234567890ab";
    fsType = "ext4";
  };

  fileSystems."/boot" = {
    device = "/dev/disk/by-uuid/1234-ABCD";
    fsType = "vfat";
  };

  swapDevices = [
    { device = "/dev/disk/by-uuid/f9e8d7c6-b5a4-3210-fedc-ba9876543210"; }
  ];

  nixpkgs.hostPlatform = lib.mkDefault "x86_64-linux";
  hardware.cpu.amd.updateMicrocode = lib.mkDefault config.hardware.enableRedistributableFirm
}
```

On a QEMU VM you will see `virtio_pci`, `virtio_blk`, `virtio_net` instead of `nvme`. That is correct. Do not copy an NVMe file onto a virtio VM.

```
blkid
ls /dev/disk/by-uuid
```

Case 2: Firmware and microcode

Save as `firmware.nix`:

```
# firmware.nix
{ config, ... }:
```

```
{
  hardware.enableRedistributableFirmware = true;
  hardware.cpu.intel.updateMicrocode = true;
  # hardware.cpu.amd.updateMicrocode = true;  # AMD hosts
}
```

Pick **one** CPU vendor. Enabling both is harmless-ish but noisy. Without microcode, you skip CPU bug fixes the vendor shipped as firmware.

Case 3: nixos-hardware for a known laptop

Save as `flake.nix` fragment (inputs + import):

```
# flake-hardware.nix
{
  description = "Desk laptop";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    nixos-hardware.url = "github:NixOS/nixos-hardware";
  };

  outputs = { self, nixpkgs, nixos-hardware }: {
    nixosConfigurations.desk-laptop = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ./configuration.nix
        ./hardware-configuration.nix
        nixos-hardware.nixosModules.framework-13-7040-amd
      ];
    };
  };
}
```

Replace the module name with the one that matches **your** chassis (`ls` the repo's module list, or the nixos-hardware README). A ThinkPad module on a Dell is worse than no module.

Case 4: Graphics — Mesa first, vendor blobs only if needed

Save as `graphics.nix`:

```
# graphics.nix
{
  hardware.graphics = {
    enable = true;
    enable32Bit = true; # Steam / wine; skip on a headless VM
```

```
  };
}
```

Intel and AMD generally need this plus firmware. NVIDIA:

```
# nvidia.nix
{
  services.xserver.videoDrivers = [ "nvidia" ];
  hardware.nvidia.open = false; # or true, per GPU generation - read the option doc
  hardware.nvidia.modesetting.enable = true;
}
```

Do not enable NVIDIA modules on a VM with virtio-gpu. The guest will load a driver for hardware that is not there.

Case 5: KVM on a workstation that runs VMs

```
# kvm.nix
{
  virtualisation.libvirtd.enable = true;
  boot.kernelModules = [ "kvm-intel" ]; # or kvm-amd
  users.users.deskadmin.extraGroups = [ "libvirtd" "kvm" ];
}
```

`boot.kernelModules` here is **host** virtualisation, not initrd disk drivers. Keep disk drivers in the generated hardware file.

```
ls /dev/kvm
```

The trap

The trap is `fileSystems."/".device = "/dev/sda2"`. Plug in a USB disk, reboot, the kernel enumerates `sda` as the USB stick, and the root UUID is now `sdb2`. Emergency shell. Always `by-uuid` or `by-id`.

The other trap is **hand-editing** `boot.initrd.availableKernelModules` out of the generated file because it "looked messy," then cloning the disk to a machine whose storage driver you deleted.

Regenerate on the new chassis (`nixos-generate-config`) and keep the host module.

The boring rule

- Generated hardware file. Your services stay in `configuration.nix`.
- Filesystems by UUID (`blkid`), never `sda`/`vda` in production configs. Disko `by-id` is the fleet version of this rule.

- Microcode + redistributable firmware on real machines.
- `nixos-hardware` only when the module name matches the chassis.
- GPU modules only for GPUs that exist.

Try this

1. `blkid` and compare UUIDs to `hardware-configuration.nix`. They must match.
2. `lsinitrd /boot/EFI/nixos/*.efi 2>/dev/null | grep -i virtio` (or `lsinitrd $(readlink /run/current-system/initrd)`) and find the disk driver.
3. `lscpu | grep -i vendor` and enable **that** vendor's `updateMicrocode`.
4. Browse `nixos-hardware` module names (`nix flake show github:NixOS/nixos-hardware` may be huge; the GitHub tree is enough). Write down the one that matches your laptop, or "none — VM."

Declarative Disk Partitioning with Disko

Declarative Disk Partitioning with Disko

`fdisk` notes in a wiki are a snowflake. The boring default is: **Disko describes GPT, LUKS, and filesystems in Nix; you format from that file; `nixos-anywhere` uses the same file on a remote disk**.

The Installing NixOS chapter partitioned a VM by hand so you would see mounts. From here on, the layout lives in git.

Mental model

Disko (`github:nix-community/disko`) is a NixOS module plus a script.

Mode	What it does
`format`	Partition, LUKS, `mkfs`. **Destroys** the disk.
`mount`	Mounts in dependency order under `/mnt`
NixOS module	Also emits `fileSystems.*` so you may not need a separate hardware file for mounts

`disk-config.nix`

```
        nixos-rebuild  →  fileSystems."/" etc.
        disko script   →  wipe /dev/disk/by-id/…  (once)
```

Device names (`/dev/sda`, `/dev/vda`, `/dev/nvme0n1`) **move**. Prefer `/dev/disk/by-id/…` on metal (`ls -l /dev/disk/by-id/` — `nvme-eui.*` or `ata-*`, not `wwn-` aliases you do not understand). On a throwaway QEMU disk, `/dev/vda` is acceptable if the VM has one disk.

`disko.devices.disk.main.device` must match **this** machine. A shared module with `device = "/dev/nvme0n1"` on a fleet is how you wipe the wrong box. Per-host `disk-config.nix`, shared *shape* (ESP size, LUKS, Btrfs `@persist`).

`nixos-anywhere` SSHes into rescue, kexecs NixOS, runs Disko, copies the closure, installs the bootloader, reboots. Same `disk-config.nix`.

Worked examples

Case 1: Lab VM — one disk, no LUKS

Save as `disk-config.nix`:

```
# disk-config.nix
{
  disko.devices.disk.main = {
    type = "disk";
    device = "/dev/vda";
    content = {
      type = "gpt";
      partitions = {
        ESP = {
          size = "1G";
          type = "EF00";
          content = {
            type = "filesystem";
            format = "vfat";
            mountpoint = "/boot";
            mountOptions = [ "umask=0077" ];
          };
        };
        root = {
          size = "100%";
          content = {
            type = "filesystem";
            format = "ext4";
            mountpoint = "/";
          };
        };
      };
    };
  };
}
```

Import it from the host:

```
# configuration.nix fragment
{ inputs, ... }:
{
  imports = [
    inputs.disko.nixosModules.disko
    ./disk-config.nix
  ];
}
```

On a **throwaway** VM that is allowed to lose `/dev/vda`:

```
sudo nix --experimental-features 'nix-command flakes' run github:nix-community/disko -- \
  --mode disko ./disk-config.nix
```

`--mode disko` formats **and** mounts. There is no undo. Not your laptop's NVMe.

Case 2: Metal — by-id + LUKS + Btrfs subvolumes

Save as `disk-luks-btrfs.nix`:

```
# disk-luks-btrfs.nix
{
  disko.devices.disk.main = {
    type = "disk";
    device = "/dev/disk/by-id/nvme-VENDOR_MODEL_SERIAL";
    content = {
      type = "gpt";
      partitions = {
        ESP = {
          size = "1G";
          type = "EF00";
          content = {
            type = "filesystem";
            format = "vfat";
            mountpoint = "/boot";
            mountOptions = [ "umask=0077" ];
          };
        };
        luks = {
          size = "100%";
          content = {
            type = "luks";
            name = "crypted";
            settings.allowDiscards = true;
            passwordFile = "/tmp/luks.key";
            content = {
              type = "btrfs";
              extraArgs = [ "-f" ];
              subvolumes = {
                "@root" = {
                  mountpoint = "/";
                  mountOptions = [ "compress=zstd" "noatime" ];
                };
                "@nix" = {
                  mountpoint = "/nix";
                  mountOptions = [ "compress=zstd" "noatime" ];
                };
                "@persist" = {
                  mountpoint = "/persist";
                  mountOptions = [ "compress=zstd" "noatime" ];
                };
                "@swap" = {
                  mountpoint = "/swap";
```

```
                swap.swapfile.size = "8G";
              };
            };
          };
        };
      };
    };
  };
}
```

`ls -l /dev/disk/by-id/` on the machine (or rescue) and paste the **id** that is not a `-partN` partition. `passwordFile` is for **provisioning**; after install, unlock is passphrase or TPM (later hardening). Do not commit `/tmp/luks.key`.

`@persist` is the impermanence contract. Create it even if you still boot a persistent root today.

Case 3: Flake input

```
# flake.nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    disko = {
      url = "github:nix-community/disko";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, disko }: {
    nixosConfigurations.desk-server = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      specialArgs = { inherit disko; };
      modules = [
        disko.nixosModules.disko
        ./disk-config.nix
        ./configuration.nix
      ];
    };
  };
}
```

```
nix eval .#nixosConfigurations.desk-server.config.fileSystems.\"/\".device
```

You should see a mapper or UUID-style device coming from Disko, not a guess.

Case 4: nixos-anywhere

```
nix run github:nix-community/nixos-anywhere -- \
  --flake .#desk-server \
  --generate-hardware-config nixos-generate-config ./hardware-configuration.nix \
  root@203.0.113.42
```

Output (shape):

```
[nixos-anywhere] Connected to rescue over SSH.
[nixos-anywhere] kexec NixOS installer...
[nixos-anywhere] Running Disko on /dev/disk/by-id/…
[nixos-anywhere] Copying closure...
[nixos-anywhere] Installing bootloader...
[nixos-anywhere] Rebooting.
```

The remote must be a rescue environment you are allowed to wipe. Hetzner rescue, a cloud "reinstall," or a live ISO with SSH. Not `root@production-that-has-data`.

Case 5: Parameterise the device per host

Save as `disk-config.nix`:

```
# disk-config.nix
{ diskDevice ? "/dev/vda", ... }:

{
  disko.devices.disk.main.device = diskDevice;
  # … rest of the layout using `main` …
}
```

Or two files: `hosts/desk-vm/disk.nix` (`/dev/vda`) and `hosts/desk-metal/disk.nix` (`by-id`). Same subvolume names so impermanence and backups stay identical.

The trap

The trap is **pointing Disko at the wrong disk** because `/dev/nvme0n1` was the installer USB on this firmware. `by-id` includes the serial. Read `lsblk -o NAME,SIZE,MODEL,SERIAL` twice.

The other trap is committing `passwordFile = "/tmp/luks.key"` **contents** into the flake, or leaving a LUKS passphrase in `settings.keyFile` as a Nix string. Same store leak as any secret.

A third trap: running `--mode format` on a machine that already has `/persist` data. Disko is an installer, not a resize tool. Back up, then format.

The boring rule

- Disk layout in git. Format from that file. Do not keep a parallel `fdisk` runbook.
- `by-id` on real disks. `/dev/vda` only on single-disk lab VMs.
- LUKS passphrase / key file is provisioning state, not a Nix string.
- Same Disko file for `nixos-anywhere` and for `fileSystems` in the running config.
- `@nix` and `@persist` as separate subvolumes before you turn on impermanence.

Try this

1. On a **disposable** QEMU VM, apply Case 1. `findmnt / /boot` after Disko mount. Confirm `/dev/vda1` is vfat and `/dev/vda2` is ext4 (names may vary).
2. `ls -l /dev/disk/by-id` on any Linux box and pick the id you would use. Do not run Disko there.
3. Eval `config.fileSystems` from a flake that imports Disko and list mountpoints.
4. Read `nixos-anywhere --help` and note `--generate-hardware-config`. That flag is how initrd modules still get generated when Disko owns mounts.

Ephemeral Roots and Impermanence

Ephemeral Roots and Impermanence

`/usr/local/bin` leftovers, an undocumented `/etc` drop-in, a rootkit in `/bin` — all of that is "state you did not declare." The boring default is: **throw / away at boot (tmpfs or a blank Btrfs snapshot), keep `/nix` and `/persist`, and list every surviving path in the `impermanence` module**.

This is optional on a laptop you are still learning on. It is the right default on a desk **server** once Disko already created `@persist` and `@nix`. Pair it with the backups chapter: git rebuilds the OS; restic rebuilds `/persist`.

Mental model

Path	Survives reboot?
`/` (tmpfs or wiped subvol)	No
`/nix`	Yes (own subvolume / partition)
`/persist`	Yes
`/boot`	Yes (ESP)
Anything not listed in `environment.persistence`	No

The `impermanence` module bind-mounts or links `/persist/…` onto the live paths (`/etc/machine-id`, `/var/log`, …) so programs still write where they always did.

```
boot
  tmpfs → /
  btrfs @nix → /nix
  btrfs @persist → /persist
  bind  /persist/etc/machine-id → /etc/machine-id
```

If you forget a path, the service fails after reboot (empty DB, new SSH host key). That failure is the audit. Add the path, or decide the data should die.

Also persist what **decrypts** secrets: `/etc/ssh/ssh_host_ed25519_key` (sops `sshKeyPaths`) and/or `/var/lib/sops-nix`. A tmpfs root plus a new host key every boot is a machine that never decrypts `secrets/desk.yaml`.

Worked examples

Case 1: tmpfs root + persist list

Save as `impermanence.nix`:

```
# impermanence.nix
{ inputs, ... }:

{
  imports = [ inputs.impermanence.nixosModules.impermanence ];

  fileSystems."/" = {
    device = "none";
    fsType = "tmpfs";
    options = [ "defaults" "size=4G" "mode=755" ];
  };

  fileSystems."/persist" = {
    neededForBoot = true;
  };

  environment.persistence."/persist" = {
    hideMounts = true;
    directories = [
      "/var/log"
      "/var/lib/nixos"
      "/var/lib/systemd/coredump"
      "/var/lib/systemd/timers"
    ];
    files = [
      "/etc/machine-id"
    ];
  };
}
```

`fileSystems."/persist"` device/fsType usually come from Disko. Do not fight Disko with a second definition of the same mount unless you are not using Disko.

`neededForBoot = true` is required so activation can see `/persist` before it tries to link `machine-id`.

Case 2: SSH host keys — the one everyone forgets

```
# persist-ssh.nix
{
  environment.persistence."/persist".files = [
    "/etc/ssh/ssh_host_ed25519_key"
    "/etc/ssh/ssh_host_ed25519_key.pub"
  ];
}
```

Without this, every reboot is a new host key. Clients print `REMOTE HOST IDENTIFICATION HAS CHANGED`. That looks like MITM. It is you.

After first boot with persistence, `ls -l /persist/etc/ssh/` should show the keys. If the files exist only under `/etc/ssh` on a still-persistent root, **copy them to `/persist`** before you enable tmpfs root, or the next boot generates new keys.

Case 3: Desk service state

```
# persist-desk.nix
{
  environment.persistence."/persist".directories = [
    "/var/lib/desk-api"
    "/var/lib/postgresql"
    "/var/lib/tailscale"
    "/var/lib/nixos"
  ];
}
```

If the unit uses `DynamicUser`, state may live under `/var/lib/private/…`. Persist **that** path or give the service a named user and a stable `StateDirectory`. Guessing wrong looks like "the database reset."

Users' homes:

```
{
  environment.persistence."/persist".users.deskadmin = {
    directories = [
      ".ssh"
      ".local/share"
    ];
    files = [
      ".bash_history"
    ];
  };
}
```

Do not persist all of `/home` "to be safe" if the point was to stop accumulating junk. Persist keys and the few tools that store tokens.

Case 4: Flake input

```
# flake.nix fragment
{
  inputs.impermanence.url = "github:nix-community/impermanence";

  outputs = { self, nixpkgs, impermanence, ... }: {
    nixosConfigurations.desk-server = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        impermanence.nixosModules.impermanence
        ./impermanence.nix
        ./configuration.nix
      ];
    };
  };
}
```

```
nix eval .#nixosConfigurations.desk-server.config.environment.persistence.\"/persist\".files
```

Output (shape):

```
[ "/etc/machine-id" "/etc/ssh/ssh_host_ed25519_key" … ]
```

If eval does not list the SSH keys, reboot will not either.

Case 5: Prove it on a VM

On a lab VM **after** persistence is on:

```
sudo touch /etc/untracked-state.txt
sudo touch /persist/i-should-survive.txt
sudo reboot
```

After boot:

```
ls /etc/untracked-state.txt
ls /persist/i-should-survive.txt
ls /etc/machine-id
```

Output:

```
ls: cannot access '/etc/untracked-state.txt': No such file or directory
/persist/i-should-survive.txt
/etc/machine-id
```

`machine-id` is back because it was listed. The untracked file is gone. If `machine-id` is also gone, `neededForBoot` or the files list is wrong — fix before you enable sshd clients.

Btrfs variant (no tmpfs): rollback `@root` to a blank snapshot in `boot.initrd.postDeviceCommands` (or the current impermanence btrfs helper). Same persist list. Prefer one mechanism; do not tmpfs **and** rollback.

The trap

The trap is **enabling tmpfs root without copying SSH host keys onto `/persist` first**. You lock yourself out over SSH. Console or a second VM disk is the recovery.

The other trap is forgetting `/var/lib/nixos` (uids for `DynamicUser` / allocated ids). After reboot, users get new uids and cannot read old state you *did* persist.

A third trap: persisting `/etc` wholesale. You are back to a mutable `/etc` with extra steps. List files.

The boring rule

- `/nix` and `/persist` (and `/boot`) are durable. `/` is not.
- List SSH host keys, `machine-id`, `/var/lib/nixos`, and each service's state dir **before** the first ephemeral boot.
- `neededForBoot = true` on `/persist`.
- Copy existing keys to `/persist` before switching root to tmpfs.
- Back up `/persist`. The flake will not grow the Postgres data back.

Try this

1. Eval the persist `files` and `directories` lists from the flake. Read them aloud. Name one path that would lock you out if missing.
2. On a **lab** VM with a snapshot you can restore, enable Case 1 + Case 2, copy host keys to `/persist`, switch, reboot, `ssh` in. Confirm the host key warning does **not** appear.
3. `sudo touch /root/junk && sudo reboot` and confirm `/root/junk` is gone.
4. Add `/var/lib/desk-api` to directories, deploy a dummy file there, reboot, confirm it survived. Remove it from the list on a second experiment and confirm it dies.

Image Generation with nixos-generators

Image Generation with nixos-generators

Every team eventually needs a bootable artifact: a rescue ISO burned to a USB stick, a QEMU image for local integration testing, an SD card for a rack of Raspberry Pis, or an AMI pushed to AWS. The boring default for all of these is **`nixos-generators`** — a flake library that accepts a standard NixOS module configuration and a format string, then produces a ready-to-use image from the Nix store. No hand-crafted `genisoimage` invocations, no bespoke `mkimage.sh` scripts, no per-format wiki pages.

Mental model

`nixos-generators` (hosted at `github:nix-community/nixos-generators`) exposes one primary function: `nixosGenerate`. It is a thin wrapper around NixOS's own `nixos-rebuild` evaluation machinery.

```
nixosGenerate {
  system  = "x86_64-linux";   # or "aarch64-linux"
  format  = "iso";            # selects the image format module
  modules = [ ./my-config.nix ];
}
```

Under the hood, `nixosGenerate` imports `<nixpkgs/nixos>` with the caller-supplied modules **plus** a format-specific module bundled inside nixos-generators. That format module enables or disables NixOS options (bootloader type, filesystem layout, network seed, cloud-init, etc.) appropriate for the target image. The result is a standard Nix derivation whose output path contains the finished image file.

Supported formats (selection):

Format string	Output file	Use case
`iso`	`.iso`	Live USB / CD (no installer scripts)
`install-iso`	`.iso`	NixOS installer (`nixos-install` / Calamares)
`qcow`	`.qcow2`	QEMU / libvirt local VMs
`amazon`	`.vhd`	AWS AMI import
`sd-aarch64`	`.img`	Raspberry Pi / ARM SBCs
`vagrant-virtualbox`	`.box`	Vagrant + VirtualBox
`do`	`.img.gz`	DigitalOcean custom images
`azure`	`.vhd`	Azure managed disk import

Key properties:

1. **Reproducible.** The same flake lock produces the same image byte-for-byte.
2. **Cacheable.** Images are regular Nix derivations; binary caches work normally.
3. **Composable.** Any NixOS module — services, users, firewall rules — works without modification.
4. **No extra tools on the build host.** All format-specific utilities (`genisoimage`, `qemu-img`, `e2fsprogs`, etc.) are pulled into the build sandbox automatically.

`format = "iso"` is a live image. Onboarding USB sticks want `install-iso` (or `modulesPath + "/installer/cd-dvd/installation-cd-minimal.nix"`). Distro name, splash, and Calamares branding are a later chapter — generators only pick the image *kind*.

Worked examples

Case 1: Bootable rescue ISO for the infrastructure desk

The team keeps a rescue ISO on a shelf USB drive. When a desk workstation fails to boot, the technician plugs in the USB and gets an SSH-accessible minimal shell with the usual diagnostic tools already present.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk rescue ISO";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nixos-generators = {
      url = "github:nix-community/nixos-generators";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nixos-generators, ... }:
    let
      system = "x86_64-linux";
      pkgs   = nixpkgs.legacyPackages.${system};
    in
    {
      packages.${system}.rescue-iso = nixos-generators.lib.nixosGenerate {
        inherit system;
        format = "iso";
        modules = [
          ({ pkgs, ... }: {
```

```
            # ISO label shown in boot menu
            isoImage.isoName = "desk-rescue-26.05.iso";

            # Make the ISO available without authentication for emergency access.
            # Restrict by placing this on a physically separate VLAN in production.
            services.openssh = {
              enable = true;
              settings.PermitRootLogin = "yes";
            };

            users.users.root.password = "rescue";

            # Diagnostic tools every technician expects
            environment.systemPackages = with pkgs; [
              vim
              curl
              htop
              tcpdump
              smartmontools
              parted
              pciutils
              usbutils
            ];

            # Stateless - no mutable filesystem needed
            system.stateVersion = "26.05";
          })
        ];
      };
    };
}
```

Build:

```
nix build .#rescue-iso
```

```
warning: Git tree '/home/ops/desk-infra' is dirty
building '/nix/store/3j7k...-nixos-rescue-iso.drv'...
/nix/store/vq84...-iso/iso/desk-rescue-26.05.iso
```

Inspect the result symlink:

```
readlink -f result
```

```
/nix/store/vq84p3rz8f2a1n06xbkj5wsywi3hqd29-iso/iso/desk-rescue-26.05.iso
```

Flash to USB:

```
sudo dd if=$(readlink -f result) of=/dev/sdX bs=4M status=progress conv=fsync
```

```
1134264320 bytes (1.1 GB, 1.1 GiB) copied, 47.3 s, 24.0 MB/s
```

Case 2: QEMU qcow2 image for local VM testing

Before deploying a new service configuration to production hosts, engineers spin up a local QEMU VM to validate it. The `qcow` format produces a copy-on-write disk image that QEMU understands natively. No conversion step is needed.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk service VM image";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nixos-generators = {
      url = "github:nix-community/nixos-generators";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nixos-generators, ... }:
    let
      system = "x86_64-linux";
      pkgs   = nixpkgs.legacyPackages.${system};
    in
    {
      packages.${system}.desk-vm = nixos-generators.lib.nixosGenerate {
        inherit system;
        format = "qcow";
        modules = [
          ({ pkgs, ... }: {
            # A minimal internal metrics collector for VM validation
            services.prometheus = {
              enable = true;
              exporters.node = {
                enable = true;
                openFirewall = true;
              };
            };
```

```
            networking.firewall.allowedTCPPorts = [ 9090 9100 ];

            # QEMU guest agent for clean shutdown via virsh/libvirt
            services.qemuGuest.enable = true;

            users.users.ops = {
              isNormalUser = true;
              extraGroups  = [ "wheel" ];
              password     = "ops";
            };

            services.openssh = {
              enable = true;
              settings.PasswordAuthentication = true;
            };

            system.stateVersion = "26.05";
          })
        ];
      };
    };
}
```

Build:

```
nix build .#desk-vm
```

```
building '/nix/store/a9xm...-desk-vm-qcow2.drv'...
/nix/store/f1yw...-nixos.qcow2
```

Launch the VM with QEMU directly:

```
qemu-system-x86_64 \
  -m 2048 \
  -smp 2 \
  -enable-kvm \
  -drive file=$(readlink -f result),format=qcow2,if=virtio \
  -net nic,model=virtio \
  -net user,hostfwd=tcp::2222-:22,hostfwd=tcp::9090-:9090 \
  -nographic
```

```
[    0.000000] Linux version 6.6.56 (nixbld@localhost) ...
[    2.148391] EXT4-fs (vda): mounted filesystem with ordered data mode.
[  OK  ] Started OpenSSH Daemon.
[  OK  ] Started Prometheus Node Exporter.
```

```
<<< NixOS 26.05 (desk-vm) >>>

desk-vm login:
```

SSH in from a second terminal:

```
ssh -p 2222 ops@127.0.0.1
```

```
[ops@desk-vm:~]$
```

Case 3: Raspberry Pi SD card image (aarch64 cross-compile from x86_64)

The team operates a small fleet of Raspberry Pi 4 boards as out-of-band management nodes on the server rack. Images must be built on a standard x86_64 workstation and flashed to SD cards.

Save as `flake.nix`:

```
# flake.nix
{
  description = "Rack OOB node SD image";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nixos-generators = {
      url = "github:nix-community/nixos-generators";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nixos-generators, ... }:
    let
      buildSystem = "x86_64-linux";
      # Cross-compilation: build on x86_64, target aarch64
      pkgsCross = nixpkgs.legacyPackages.${buildSystem}.pkgsCross.aarch64-multiplatform;
    in
    {
      packages.${buildSystem}.oob-sd = nixos-generators.lib.nixosGenerate {
        # The *host* (target board) architecture
        system  = "aarch64-linux";
        # Supply pkgs built via the cross toolchain so no aarch64 runners are needed
        pkgs    = pkgsCross;
        format  = "sd-aarch64";
        modules = [
          ({ pkgs, lib, ... }: {
            # Raspberry Pi 4B hardware support
            hardware.raspberry-pi."4".apply-overlays-dtmerge.enable = true;
```

```
            hardware.deviceTree.enable = true;

            networking.hostName = "oob-node";

            services.openssh = {
              enable = true;
              settings = {
                PermitRootLogin        = "no";
                PasswordAuthentication = false;
              };
            };

            users.users.ops = {
              isNormalUser    = true;
              extraGroups     = [ "wheel" ];
              openssh.authorizedKeys.keys = [
                "ssh-ed25519 AAAA... ops@desk"
              ];
            };

            security.sudo.wheelNeedsPassword = false;

            environment.systemPackages = with pkgs; [
              vim
              curl
              iproute2
              iputils
            ];

            system.stateVersion = "26.05";
          })
        ];
      };
    };
}
```

Build (cross-compilation; no QEMU emulation required):

```
nix build .#oob-sd
```

```
building '/nix/store/c2hm...-sd-image-aarch64-linux.drv'...
/nix/store/p9bw...-nixos-sd-image-26.05.aarch64-linux.img
```

Find the image path and flash it:

```
SD_IMG=$(readlink -f result)

sudo dd if="$SD_IMG" of=/dev/mmcblk0 bs=4M status=progress conv=fsync
```

```
3936280576 bytes (3.9 GB, 3.7 GiB) copied, 124 s, 31.7 MB/s
```

Eject the card, insert it into the Pi, and power on. The node appears on the rack management VLAN within 30 seconds.

Case 4: Sharing a NixOS module between a live machine and an image build

The biggest maintenance risk when maintaining images is drift: the image diverges from the live system because they evolved separately. The fix is to extract shared configuration into a standalone module file and import it in both the `nixosConfigurations` output and the `nixosGenerate` call.

Save as `modules/desk-base.nix`:

```
# modules/desk-base.nix
{ pkgs, lib, ... }:
{
  # Shared by both live desk machines and generated images

  networking.domain = "ops.example.internal";

  services.openssh = {
    enable = true;
    settings = {
      PermitRootLogin        = "no";
      PasswordAuthentication = false;
      X11Forwarding          = false;
    };
  };

  environment.systemPackages = with pkgs; [
    vim
    git
    curl
    htop
    jq
  ];

  security.sudo.wheelNeedsPassword = true;

  time.timeZone = "UTC";

  nix = {
```

```
    settings.experimental-features = [ "nix-command" "flakes" ];
    gc = {
      automatic = true;
      dates     = "weekly";
      options   = "--delete-older-than 30d";
    };
  };

  system.stateVersion = "26.05";
}
```

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk infrastructure - shared module demo";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

    nixos-generators = {
      url = "github:nix-community/nixos-generators";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, nixos-generators, ... }:
    let
      system = "x86_64-linux";
      pkgs   = nixpkgs.legacyPackages.${system};

      # The shared module - imported in both outputs below
      deskBase = ./modules/desk-base.nix;
    in
    {
      #   Live system deployed to real hardware
      nixosConfigurations.desk-01 = nixpkgs.lib.nixosSystem {
        inherit system;
        modules = [
          deskBase
          ./hosts/desk-01/hardware-configuration.nix
          ({ ... }: {
            networking.hostName = "desk-01";
            users.users.alice = {
              isNormalUser = true;
              extraGroups  = [ "wheel" ];
            };
```

```
          })
        ];
      };

      #   ISO image for emergency rescue of desk machines
      packages.${system}.desk-rescue-iso = nixos-generators.lib.nixosGenerate {
        inherit system;
        format = "iso";
        modules = [
          # Same shared module → same tools, same SSH policy, same Nix config
          deskBase
          ({ lib, ... }: {
            networking.hostName = "desk-rescue";

            # Rescue-only override: allow root login for emergency access
            services.openssh.settings.PermitRootLogin = lib.mkForce "yes";
            users.users.root.password = "rescue";

            isoImage.isoName = "desk-rescue-26.05.iso";
          })
        ];
      };

      #   QEMU image for pre-deploy validation
      packages.${system}.desk-vm = nixos-generators.lib.nixosGenerate {
        inherit system;
        format = "qcow";
        modules = [
          deskBase
          ({ lib, ... }: {
            networking.hostName = "desk-vm";
            services.qemuGuest.enable = true;
            users.users.root.password = "vm";
            services.openssh.settings.PermitRootLogin = lib.mkForce "yes";
          })
        ];
      };
    };
}
```

Build both images without touching `modules/desk-base.nix`:

```
nix build .#desk-rescue-iso
```

`/nix/store/8k3n...-iso/iso/desk-rescue-26.05.iso`

```
nix build .#desk-vm
```

```
/nix/store/r7cj...-nixos.qcow2
```

Any change to `modules/desk-base.nix` — adding a package, tightening an SSH setting — is reflected in **both** the live system and every image output on the next build. There is no separate image-maintenance checklist.

Case 5: Minimizing image closures for lightweight appliances

A standard NixOS image often ships with documentation, desktop hooks, and interpreters that a headless appliance or edge node does not need. The image can inflate from 300 MB to over 1.5 GB without any obvious additions in your configuration.

Measure the heaviest packages in the image closure:

```
nix-store --query --requisites $(nix build --print-out-paths .#packages.x86_64-linux.desk-ap
  | xargs du -sm | sort -n | tail -n 10
```

Sample output:

```
43  /nix/store/...-systemd-257.9
63  /nix/store/...-perl-5.40.0
114 /nix/store/...-python3-3.13.7
144 /nix/store/...-linux-6.12.53-modules
229 /nix/store/...-mesa-25.2.5
541 /nix/store/...-llvm-21.1.1-lib
648 /nix/store/...-source
```

Trace the unexpected dependencies with `nix why-depends`:

```
nix why-depends $(nix build --print-out-paths .#packages.x86_64-linux.desk-appliance) \
  /nix/store/...-llvm-21.1.1-lib
```

Save optimization settings in `modules/appliance-minimal.nix`:

```
# modules/appliance-minimal.nix
{ lib, pkgs, modulesPath, ... }:

{
  # 1. Eliminate Perl completely from the image
  imports = [ (modulesPath + "/profiles/perlless.nix") ];

  # Assert that Perl never leaks back into the closure
  system.forbiddenDependenciesRegexes = [ "perl" ];
```

```
  # 2. Disable stealth subsystems
  # speechd pulls mbrola which includes 650MB+ of voice sources
  services.speechd.enable = false;

  # Mesa + LLVM are unnecessary on headless appliances
  hardware.graphics.enable = false;

  # Audio and input stacks that drag in Python via gstreamer
  services.pipewire.enable = false;
  services.libinput.enable = false;

  # 3. Strip font catalogs
  fonts.enableDefaultPackages = false;
  fonts.fontconfig.enable = false;
  fonts.packages = lib.mkForce [ pkgs.dejavu_fonts ];

  # 4. Stub hardcoded upstream tools via overlays
  nixpkgs.overlays = [
    (final: prev: {
      # Stub xdg-utils to bash when a desktop module hardcodes it
      xdg-utils = final.bash;
    })
  ];
}
```

Import `modules/appliance-minimal.nix` into your image definition. The closure drops from ~1.5 GB to ~360 MB, cutting download times and attack surface.

The trap

Building `sd-aarch64` on x86_64 without cross-compilation

The `sd-aarch64` format targets `aarch64-linux`. If you forget to supply a cross-compiled `pkgs` and simply pass the host `pkgs`, Nix will attempt to run aarch64 ELF binaries inside the build sandbox on an x86_64 kernel:

```
# flake.nix  (wrong - do not use)
packages.x86_64-linux.bad-sd = nixos-generators.lib.nixosGenerate {
  system = "aarch64-linux";
  # No pkgs override - falls back to x86_64 pkgs, then tries to exec aarch64
  format  = "sd-aarch64";
  modules = [ ./modules/desk-base.nix ];
};
```

Build error:

```
building '/nix/store/...-bash-5.2-aarch64-unknown-linux-gnu.drv'...
error: builder for '/nix/store/...-bash-5.2-aarch64-unknown-linux-gnu.drv' failed:
  error: executing '/nix/store/...-bash-5.2/bin/bash':
  Exec format error
```

The kernel refuses to execute an aarch64 ELF. There are two correct fixes.

Fix A — Cross-compilation (no emulation needed, recommended):

```
# flake.nix  (relevant excerpt)
let
  buildSystem = "x86_64-linux";
  pkgsCross   = nixpkgs.legacyPackages.${buildSystem}.pkgsCross.aarch64-multiplatform;
in
{
  packages.${buildSystem}.correct-sd = nixos-generators.lib.nixosGenerate {
    system = "aarch64-linux";
    pkgs   = pkgsCross;           # cross-compiled pkgs supplied explicitly
    format = "sd-aarch64";
    modules = [ ./modules/desk-base.nix ];
  };
}
```

Fix B — binfmt emulation (slower, requires a NixOS build host):

Add this to your build host's `configuration.nix`, then `nixos-rebuild switch`:

```
# configuration.nix  (build host only)
{ ... }:
{
  boot.binfmt.emulatedSystems = [ "aarch64-linux" ];
}
```

```
nixos-rebuild switch
```

```
activating the configuration...
setting up binfmt handlers...
registering aarch64-linux as binfmt_misc handler
```

With binfmt active, aarch64 ELFs are transparently executed under QEMU user-mode emulation. Builds succeed but take significantly longer than cross-compilation. Use Fix A for routine CI pipelines; Fix B only when cross-compilation is not available.

The boring rule

- **Use `nixos-generators` instead of custom image scripts.** All format-specific tooling is inside the Nix sandbox; your repository stays clean.

- **Share modules between live systems and image builds.** Put common configuration in `modules/` files and import them in both `nixosConfigurations` and `nixosGenerate` calls. Images reflect exactly what runs in production.
- **Always pin `nixos-generators` in `flake.lock`.** Run `nix flake update nixos-generators` deliberately, not on every build. Image reproducibility depends on a stable lock.
- **Use `pkgsCross.aarch64-multiplatform` for cross-architecture builds.** Do not rely on binfmt emulation in CI; it is slower and silently fails on kernels without the `binfmt_misc` module loaded.
- **Audit and minimize appliance closures.** Measure with `nix-store --query --requisites`, blame with `why-depends`, strip stealth subsystems (`speechd`, `graphics`), and enforce constraints with `system.forbiddenDependenciesRegexes`.
- **Name image outputs descriptively** (`rescue-iso`, `desk-vm`, `oob-sd`) so `nix build .#<tab>` is self-documenting.

Try this

1. **Extend the rescue ISO.** Add `nmap` and `wireguard-tools` to the `rescue-iso` packages list and rebuild. Verify both binaries appear inside the ISO by mounting it: `sudo mount -o loop result /mnt` followed by `ls /mnt/nix/store | grep -E "nmap|wireguard"`.

2. **Boot the QEMU image with a persistent overlay.** Create a 4 GB writable overlay with `qemu-img create -f qcow2 -b $(readlink -f result) -F qcow2 overlay.qcow2 4G`, then boot it with the overlay path. Confirm that writes survive a reboot while the base image stays unchanged.

3. **Add a DigitalOcean image format without duplicating config.** In the Case 4 flake, add a `packages.x86_64-linux.desk-do` output using `format = "do"` and the same `deskBase` module. Build it with `nix build .#desk-do` and inspect the resulting `.img.gz`.

4. **Measure image closure size.** After building any image, run `nix path-info -rS $(readlink -f result) | sort -k2 -rn | head -20` to identify the largest packages in the closure. Remove one unnecessary package from the module and compare closure sizes before and after.

5. **Profile and minimize an appliance image.** Import `(modulesPath + "/profiles/perlless.nix")` and set `system.forbiddenDependenciesRegexes = [ "perl" ]`. Verify with `nix build` that the output builds cleanly without Perl in the runtime closure.

Backups and Generation Hygiene

Backups and Generation Hygiene

A flake rebuilds the **operating system**. It does not recreate Postgres, `/persist`, or the Age key you generated once. The boring default is: **restic (or borg) of state, a restore drill, and GC that keeps two weeks of generations — not `rm -rf /nix`**.

Mental model

Rebuild from git + cache	Must be backed up
`/nix/store`	`/persist` (impermanence)
Kernel, units, packages	Database files / dumps
Most of `/etc`	Secrets material you cannot reissue
Home Manager **managed** files	Unmanaged `$HOME` (photos, SSH private keys if not in sops)

3-2-1 still applies: live + backup + offsite. A green `restic snapshots` line without a restore is theatre.

Generation hygiene is the other disk: old NixOS generations are rollback insurance **and** gigabytes. Keep a window. Automatic GC with `--delete-older-than 14d` is the boring timer. `-d` is for incidents.

```
wipe disk
     nixos-anywhere / installer  →  OS from flake
     restic restore              →  /persist + DB
```

Worked examples

Case 1: What to include

Save as `backup-paths.md` in the desk repo (documentation, not Nix):

```
# backup-paths.md
Include:
  /persist
  /var/lib/postgresql
  /var/lib/desk-api
```

```
Exclude:
  /nix/store
  /tmp
  /var/cache
  result
  .direnv
```

If you use impermanence, **`/persist` is the list**. Do not also back up the ephemeral root.

Case 2: Restic on NixOS

Save as `restic.nix`:

```nix
# restic.nix
{ config, pkgs, ... }:

{
  services.restic.backups.desk = {
    initialize = true;
    passwordFile = config.sops.secrets.restic-password.path;
    repository = "sftp:backup@backup.desk.internal:/backups/desk-vm";
    paths = [
      "/persist"
      "/var/lib/desk-api"
    ];
    pruneOpts = [
      "--keep-daily 7"
      "--keep-weekly 4"
      "--keep-monthly 6"
    ];
    timerConfig = {
      OnCalendar = "daily";
      Persistent = true;
    };
  };
}
```

The repository password is a sops secret, not a string in this file. `sftp:` is one backend; `s3:s3.amazonaws.com/desk-backups` is the same module.

```nix
# restic.nix extras
{ config, pkgs, ... }:

{
  services.restic.backups.desk = {
    user = "root"; # default. pg_dumpall needs to become postgres.
    environmentFile = config.sops.secrets.restic-env.path; # AWS keys if s3:
```

```
    extraBackupArgs = [ "--exclude-caches" "--tag" "desk" ];
    exclude = [ "/persist/backups/*.tmp" ];
    backupPrepareCommand = ''
      set -euo pipefail
      mkdir -p /persist/backups
      ${pkgs.sudo}/bin/sudo -u postgres \
        ${config.services.postgresql.package}/bin/pg_dumpall -c \
        -f /persist/backups/pg.dump
    '';
    backupCleanupCommand = ''
      rm -f /persist/backups/pg.dump
    '';
    runCheck = true;
  };
}
```

Dump **before** restic walks `/persist`. A running cluster's data files without a dump are a crash-consistent maybe. `backupPrepareCommand` is a script NixOS wraps with `pkgs.writeScript` — interpolate **store paths** (`config.services.postgresql.package`, not `pg_dump` from PATH). `set -euo pipefail` so a failed dump fails the unit instead of uploading yesterday's file.

`user` defaults to `"root"`. If you set `user = "restic"`, `sudo -u postgres` needs a sudoers rule, or skip sudo and `user = "postgres"` **only** when the paths are that uid's. Do not invent a wrapper until the default root unit is proven.

`backupCleanupCommand` removes the dump from disk after the snapshot (the dump still lives **in** the snapshot). `runCheck` is `restic check` after prune — catch a silent repo corruption on the timer, not at restore time.

`createWrapper = true` (module default on many pins) installs a `restic-desk` wrapper with the same env as the unit. Humans restore with that, not a bare `restic` that is missing `RESTIC_PASSWORD_FILE`.

```
sudo nixos-rebuild switch
sudo systemctl start restic-backups-desk.service
sudo journalctl -u restic-backups-desk.service -n 40
```

Case 3: Restore drill (the part people skip)

```
sudo restic -r sftp:backup@backup.desk.internal:/backups/desk-vm \
  --password-file /run/secrets/restic-password \
  snapshots

sudo restic -r sftp:backup@backup.desk.internal:/backups/desk-vm \
  --password-file /run/secrets/restic-password \
  restore latest --target /tmp/desk-restore-drill
```

```
ls /tmp/desk-restore-drill/persist | head
```

On a lab VM, go further: restore `/var/lib/desk-api` onto a **second** VM and start the service. A file listing is not a drill. A responding API is.

Schedule this quarterly. Write the date in the desk repo.

Case 4: Generation window on NixOS

Save as `gc.nix`:

```
# gc.nix
{
  nix.gc = {
    automatic = true;
    dates = "weekly";
    options = "--delete-older-than 14d";
    persistent = true; # catch up after the box was off
  };
  nix.optimise.automatic = true;
  boot.loader.systemd-boot.configurationLimit = 8;
}
```

`configurationLimit` caps bootloader entries so `/boot` (often a small ESP) does not fill. GC and the bootloader limit work together: deleting generations without limiting entries leaves stale boot menu rows; limiting entries without GC leaves the store fat.

```
sudo nixos-rebuild switch
sudo nix-env --list-generations --profile /nix/var/nix/profiles/system
```

Case 5: `/boot` full is a different incident

```
df -h /boot
sudo nixos-rebuild switch
```

If `/boot` is 100%, `switch` fails while `/nix` still has space. `configurationLimit` plus removing old entries:

```
sudo nix-collect-garbage --delete-older-than 7d
sudo nixos-rebuild switch
```

Do not `rm /boot/EFI/nixos/*` by hand unless you know which generation you are running.

The trap

The trap is **backing up `/nix/store`**. It is huge, it is reconstructable, and restic will spend all night hashing NARs. Back up state. Substitute the store from the flake + cache.

The other trap is `nix-collect-garbage -d` on Friday before a Monday upgrade. Keep 14 days. Use `-d` when `df` is the outage.

A third trap: restic password in `repository` URL query string inside `configuration.nix`. That string is in the world-readable store.

A fourth: `backupPrepareCommand = "pg_dumpall …"` with no store path — the unit's PATH is not your login shell. A fifth: restoring the dump **over** a live cluster that already `initdb`'d empty (DR chapter: `--no-reboot`, restore `/persist` first).

The boring rule

- Flake + cache = OS. Restic (or borg) = state. You need both.
- `pg_dumpall` via store-path `backupPrepareCommand` before the walk. `runCheck` after.
- Restore drill with a second VM (empty Disko), not only `restic snapshots`.
- Weekly GC with `--delete-older-than 14d`. `configurationLimit` on the ESP.
- Never back up `/nix/store` as the backup strategy.
- Backup passwords via sops, not Nix strings. Use the generated wrapper.

Try this

1. Inventory `/persist` (or `/var/lib`) on a lab VM and write the include/exclude list.
2. Enable Case 4, `systemctl status nix-gc.timer`, and list generations.
3. Fill a dummy `/persist/desk-note.txt`, take a restic snapshot to a local `repository = "/var/backup/restic";` (lab), delete the file, restore, confirm the text.
4. Set `configurationLimit = 2` on a VM, switch twice, reboot, and count systemd-boot entries. Put it back to 8.
5. `systemctl cat restic-backups-desk.service` and confirm `pg_dumpall` is a `/nix/store/…-postgresql-…` path.

Systemd Stage 1 Initrd

Systemd Stage 1 Initrd

NixOS 26.05 made **systemd Stage 1** the default initrd. The old busybox/scripted Stage 1 is deprecated and scheduled for removal in 26.11. The boring default is: **leave `boot.initrd.systemd.enable` on (the 26.05 default), point root at `/dev/mapper/<name>` for LUKS, express early boot work as `boot.initrd.systemd.services` / `.mounts`, and debug with `rd.systemd.debug_shell` — do not flip the switch back to scripted Stage 1 to "make it work."**

Stage 2 systemd (the services chapter) already runs after pivot-root. Stage 1 is the same language, earlier: unlock disks, mount `/sysroot`, run oneshots, then hand off.

Mental model

```
firmware / UKI / systemd-boot
```

```
  Stage 1  (initrd)
  systemd PID 1 in the initrd
  cryptsetup · mounts · fsck
  custom oneshots
  sysroot.mount → switch-root
```

```
  Stage 2  (real root)
  systemd.services.*
  multi-user.target
```

Concern	Scripted Stage 1 (legacy)	Systemd Stage 1 (26.05 default)
Orchestration	Shell snippets, `preLVMCommands`, order by string concat	Units, generators, `RequiresMountsFor=`
LUKS	`cryptsetup` in a script	`systemd-cryptsetup@` from `boot.initrd.luks.devices`
Key file on a filesystem	Mount by hand in a script	`boot.initrd.systemd.mounts` + automatic deps

Concern	Scripted Stage 1 (legacy)	Systemd Stage 1 (26.05 default)
FIDO2 / TPM2 unlock	Awkward / third-party	`crypttabExtraOpts` + `systemd-cryptenroll`
Emergency shell	Busybox ash tricks	`systemctl`, `journalctl`, `rd.systemd.debug_shell`
Root in initrd	`/mnt-root`	`/sysroot`

On 26.05 you rarely set `boot.initrd.systemd.enable = true;` — it is already true. Explicit `= true` in snippets below is documentation, not a requirement for new hosts. `= false` is a 26.11 landmine. Debug with `rd.systemd.debug_shell` on the kernel cmdline (Stage 1 `tty`), not by reverting to scripted initrd.

LUKS mapped names in Stage 1 are `/dev/mapper/<name>` from `boot.initrd.luks.devices.<name>`. Disko's LUKS name must match that `<name>` or `sysroot.mount` waits forever. Root is `/sysroot` in the initrd, `/` after switch-root — a custom oneshot that writes `/etc` in Stage 1 is writing the initrd overlay, not persist.

Worked examples

Case 1: Confirm Stage 1 is systemd on a desk workstation

Save as `initrd_probe.nix` and import it from the host flake:

```
# initrd_probe.nix
{ lib, ... }:

{
  # Explicit for clarity on a 26.05 desk host. Omit on greenfield configs.
  boot.initrd.systemd.enable = true;

  # Keep a few generations so a bad initrd is recoverable from the boot menu.
  boot.loader.systemd-boot.configurationLimit = 8;
}
```

After `nixos-rebuild switch` (and a reboot if you just upgraded from 25.11):

```
# Is PID 1 in the initrd systemd? Check the running system metadata:
systemctl --version | head -n1
lsinitrd /run/current-system/initrd 2>/dev/null | rg -i 'systemd|initrd' | head
cat /run/current-system/kernel-params
```

Representative output on a healthy 26.05 desk host:

```
systemd 257 (257.x)
```

```
...
drwxr-xr-x   1 root     root            0 Jan  1 00:00 etc/systemd
...
init=/nix/store/…-systemd-stage-1-init/init
```

If you still see `boot.initrd.systemd.enable = false` somewhere in the merge, find it:

```
nixos-option boot.initrd.systemd.enable
```

```
Value:
true

Default:
true
```

Case 2: LUKS root that does not time out

The #1 upgrade breakage: `fileSystems."/".device` still points at the raw disk (or a by-uuid of the **encrypted** partition) while systemd Stage 1 waits for a device unit that never becomes the mounted root. Point `/` at the **mapper** name that matches `boot.initrd.luks.devices.<name>`.

Save as `luks_root.nix`:

```
# luks_root.nix
{ lib, ... }:

{
  boot.initrd.systemd.enable = true;

  boot.initrd.luks.devices.cryptroot = {
    device = "/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d";
    # Optional: allow FIDO2 tokens enrolled with systemd-cryptenroll.
    # crypttabExtraOpts = [ "fido2-device=auto" ];
  };

  fileSystems."/" = {
    device = "/dev/mapper/cryptroot";
    fsType = "ext4";
  };

  fileSystems."/boot" = {
    device = "/dev/disk/by-uuid/ABCD-1234";
    fsType = "vfat";
    options = [ "umask=0077" ];
  };

  # Complex stacks (LVM on LUKS, slow USB keyfiles): never time out the root device.
```

```
  # Prefer fixing the mapper path first; use infinity only when the topology needs it.
  # fileSystems."/".options = [ "x-systemd.device-timeout=infinity" ];
}
```

Build without activating (safe on a laptop that is not this host):

```
nixos-rebuild build --flake .#desk-workstation
```

```
building the system configuration...
/nix/store/xxxxxxxx-nixos-system-desk-workstation-26.05
```

Emergency kernel cmdline if you already cannot boot (from the bootloader editor):

```
systemd.default_device_timeout_sec=infinity
```

Then fix `fileSystems."/".device` and rebuild.

Case 3: Keyfile on a separate filesystem — mounts, not sleep scripts

Scripted Stage 1 taught people to `sleep 2; mount …` in `boot.initrd.systemd.services`. With systemd Stage 1, declare the mount. `systemd-cryptsetup` grows a `RequiresMountsFor=` on the key path automatically.

Save as `luks_keyfile_mount.nix`:

```
# luks_keyfile_mount.nix
{
  boot.initrd.systemd.enable = true;

  boot.initrd.systemd.mounts = [
    {
      what = "UUID=b501f1b9-7714-472c-988f-3c997f146a17";
      where = "/key";
      type = "ext4";
      options = "ro";
    }
  ];

  boot.initrd.luks.devices.cryptroot = {
    device = "/dev/disk/by-uuid/3f6b0024-3a44-4fde-a43a-767b872abe5d";
    keyFile = "/key/root.key";
  };

  fileSystems."/" = {
    device = "/dev/mapper/cryptroot";
    fsType = "btrfs";
```

```
    options = [ "subvol=@" ];
  };
}
```

Dependency chain systemd builds for you:

```
UUID=… device appears
   → key.mount
        → systemd-cryptsetup@cryptroot.service (RequiresMountsFor=/key/root.key)
             → sysroot.mount (/dev/mapper/cryptroot)
                  → initrd.target → switch-root
```

Do **not** put `/mnt-root` in a key path. Under systemd Stage 1 the real-root mount is `/sysroot`.

Case 4: A Stage 1 oneshot for the desk (impermanence-friendly)

Ephemeral-root desks often need a tiny action before `sysroot` is writable enough for Stage 2 — for example, ensuring a Btrfs subvolume exists, or wiping `/` except persisted paths. Express it as an initrd service with `DefaultDependencies = "no"` and explicit ordering.

Save as `initrd_wipe_root.nix`:

```
# initrd_wipe_root.nix
{ pkgs, ... }:

{
  boot.initrd.systemd.enable = true;

  boot.initrd.systemd.services.desk-prepare-root = {
    description = "Desk: prepare ephemeral root before switch-root";
    wantedBy = [ "initrd.target" ];
    before = [ "sysroot.mount" ];
    unitConfig.DefaultDependencies = "no";
    serviceConfig.Type = "oneshot";
    # Store paths only - no ambient PATH in Stage 1.
    path = [ pkgs.coreutils pkgs.btrfs-progs ];
    script = ''
      set -eu
      # Illustrative: create a subvolume the Stage 2 impermanence module expects.
      # Real desks wire this to the same layout as the Disko / impermanence chapters.
      if ! btrfs subvolume show /sysroot/@ 2>/dev/null; then
        echo "desk-prepare-root: @ missing - check Disko layout"
      fi
    '';
  };
}
```

Inspect whether the unit is in the system closure after a dry build:

```
nixos-rebuild build --flake .#desk-workstation
nix-store -q --references result | rg -i 'initrd|systemd' | head
```

While debugging a live Stage 1 (see Case 5), `systemctl status desk-prepare-root.service` shows the oneshot before switch-root. After switch-root those units are gone — that is expected.

Case 5: Debug a stuck Stage 1 with the emergency shell

When `sysroot.mount` fails, you need a shell **inside** the initrd, not Stage 2 `rescue.target`.

1. At the systemd-boot (or GRUB) editor, append:

`rd.systemd.debug_shell`

2. Boot. A root shell on tty9 (often Alt+F9) appears while Stage 1 is running.
3. Useful commands in that shell:

```
systemctl
systemctl status sysroot.mount
systemctl status systemd-cryptsetup@cryptroot.service
journalctl -b -u sysroot.mount
systemctl default          # retry reaching initrd.target; also runs ask-password agent
```

Save a flake fragment that keeps the debug shell available without typing cmdline every time **on a lab VM only**:

```
# initrd_debug_lab.nix
{
  boot.initrd.systemd.enable = true;
  # Lab VMs only - this is a root shell before the root filesystem is trusted.
  boot.kernelParams = [ "rd.systemd.debug_shell" ];
}
```

Never leave `rd.systemd.debug_shell` on a production desk host that sits in an unlocked room.

The trap

Trap A — flip `boot.initrd.systemd.enable = false` to silence a LUKS timeout. That restores the deprecated scripted initrd. It will be removed in 26.11. You then maintain two boot stories and break FIDO2/`systemd-cryptenroll` paths. Fix the mapper device and mount units instead.

Trap B — keep `postDeviceCommands` / `preLVMCommands` / `/mnt-root` assumptions. Those are scripted-Stage-1 APIs. Under systemd Stage 1, early work is units; the root mount is `/sysroot`. Copied blog snippets from 23.11 often fail silently or race.

Trap C — keyfile = `/dev/mapper/usb` as a raw device. Newer systemd crypttab behaviour expects a filesystem+path form (`/key:/dev/mapper/usb`) or a real mount unit + file path. Raw-device keyfiles that "worked" under patched older NixOS can stall after upgrade.

```
# Anti-pattern - do not "fix" Stage 1 by reverting
{
  boot.initrd.systemd.enable = false;  # deprecated; dies in 26.11
  boot.initrd.postDeviceCommands = ''
    sleep 2
    …
  '';
}
```

```
# Boring fix - stay on systemd Stage 1
{
  boot.initrd.luks.devices.cryptroot.device = "/dev/disk/by-uuid/…";
  fileSystems."/".device = "/dev/mapper/cryptroot";
  boot.initrd.systemd.mounts = [ { what = "UUID=…"; where = "/key"; type = "ext4"; } ];
}
```

The boring rule

- Keep systemd Stage 1 enabled on NixOS 26.05+. Treat `enable = false` as a temporary incident control, not a config style.
- LUKS root: `boot.initrd.luks.devices.<name>` **and** `fileSystems."/".device = "/dev/mapper/<name>"`.
- Key material on another filesystem: `boot.initrd.systemd.mounts`, not `sleep` + `mount` oneshots.
- Custom early boot: `boot.initrd.systemd.services` with `DefaultDependencies = "no"` and `before = [ "sysroot.mount" ];`.
- Paths: `/sysroot`, never `/mnt-root`, in Stage 1 scripts.
- Debug with `rd.systemd.debug_shell` and `systemctl status`; fix units; then remove the debug cmdline.
- Disko still owns partitioning; this chapter owns how Stage 1 **opens** what Disko created.

Try this

1. On a disposable 26.05 VM, run `nixos-option boot.initrd.systemd.enable` and confirm `true`. Force `false`, rebuild, note the eval/boot warnings, then set it back.
2. Take the Disko LUKS layout from the Disko chapter. Verify `fileSystems."/".device` is `/dev/mapper/…` matching the LUKS name. Mis-point it at the raw by-uuid, `nixos-rebuild build-vm`, and watch Stage 1 complain in the serial log.
3. Add a `boot.initrd.systemd.mounts` entry for a second disk labeled `KEYS` and unlock with `keyFile = "/key/root.key"`. Confirm order with unit status from a failed `rd.systemd.debug_shell` session.

4. Enroll a FIDO2 token with `systemd-cryptenroll --fido2-device=auto /dev/disk/by-uuid/…`, set `crypttabExtraOpts = [ "fido2-device=auto" ];`, and boot once with the token present and once without (expect passphrase fallback if a passphrase slot remains).
5. Schedule the removal: grep the desk flake for `postDeviceCommands`, `preLVMCommands`, and `boot.initrd.systemd.enable = false`. Open tickets for each hit before 26.11.

Custom NixOS ISOs and Distro Branding

Custom NixOS ISOs and Distro Branding

A rescue ISO with a renamed file is not a distro. The boring default is: **import the official installer CD module, put identity and artwork in one shared module, and install from that ISO into the same flake — do not fork nixpkgs to change a logo**.

The generators chapter builds images (ISO, qcow, SD). This chapter is the **installer / live distro**: boot menu, `/etc/os-release`, Plymouth, greeter, Calamares. The desk ships **DeskOS** internally — NixOS 26.05 with a name, colours, and a first-boot story the onboarding laptop already trusts.

NixOS logos and the snowflake are a **separate brand**. Do not recolour them and call the result yours. Use your own mark. Keep `ID_LIKE=nixos` so tools still know what they are on.

Mental model

Layer	What a human sees	Nix
File name / USB label	`DeskOS-26.05.iso`, volume in `lsblk`	`isoImage.isoName`, `volumeID`, `edition`
Boot menu	Title, background, GRUB theme	`distroName`, splash images, `grubTheme`, `syslinuxTheme`
Live OS	`hostnamectl`, `cat /etc/os-release`	`system.nixos.distroId` / `distroName` / `extraOSReleaseArgs`
Boot splash	Logo while the kernel starts	`boot.plymouth`
Graphical installer	Calamares window, slideshow	vendored `branding.desc` + PNGs
Installed machine	Same name after `nixos-install`	**same identity module** in `nixosConfigurations`

```
artwork/  +  modules/desk-identity.nix


  installer ISO   live USB   desk-laptop (flake)
```

`iso` (live) `install-iso` (has `nixos-install` / Calamares). Pick the installer module on purpose.

ISO9660 `volumeID` is short (keep it under 32 characters). A 60-character marketing name belongs in `isoName` and `PRETTY_NAME`, not on the volume.

Worked examples

Case 1: Minimal installer ISO from `modulesPath`

Save as `flake.nix`:

```
# flake.nix
{
  description = "DeskOS installer ISO";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.desk-iso = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ({ modulesPath, pkgs, lib, ... }: {
          imports = [
            (modulesPath + "/installer/cd-dvd/installation-cd-minimal.nix")
          ];

          isoImage.isoName = "DeskOS-26.05-x86_64.iso";
          isoImage.edition = "desk";
          isoImage.volumeID = "DESKOS-2605-X64";
          isoImage.makeEfiBootable = true;
          isoImage.makeUsbBootable = true;
          isoImage.appendToMenuLabel = " installer";

          environment.systemPackages = with pkgs; [
            vim git curl jq
            disko
          ];

          nix.settings.experimental-features = [ "nix-command" "flakes" ];

          services.openssh.enable = true;
          users.users.nixos.openssh.authorizedKeys.keys = [
            "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI… deskadmin@ops-01"
          ];

          system.stateVersion = "26.05";
        })
      ];
    };
  };
}
```

```
nix build .#nixosConfigurations.desk-iso.config.system.build.isoImage
ls -lh result/iso/
```

Output (shape):

```
-r--r--r-- 1 root root 1.2G  DeskOS-26.05-x86_64.iso
```

On 26.05 you can also:

```
nixos-rebuild build-image --image-variant iso-installer --flake .#desk-iso
```

If the flake already has `nixos-generators`, `format = "install-iso"` is the same installer family. `format = "iso"` is a live image **without** the installer scripts — wrong for onboarding.

Flash:

```
sudo dd if=$(readlink -f result/iso/*.iso) of=/dev/sdX bs=4M status=progress conv=fsync
```

Case 2: Identity — `/etc/os-release`, not a wallpaper

Save as `modules/desk-identity.nix`:

```
# modules/desk-identity.nix
{ lib, ... }:
{
  system.nixos.distroId = "deskos";
  system.nixos.distroName = "DeskOS";
  system.nixos.vendorId = "desk";
  system.nixos.vendorName = "Desk Platform";
  system.nixos.variant_id = "workstation";
  system.nixos.variantName = "DeskOS Workstation";
  system.image.id = "deskos-installer";
  system.image.version = "26.05";

  # When distroId != "nixos", NixOS already sets ID_LIKE=nixos.
  system.nixos.extraOSReleaseArgs = {
    HOME_URL = "https://git.desk.internal/platform/deskos";
    DOCUMENTATION_URL = "https://git.desk.internal/platform/deskos#readme";
    SUPPORT_URL = "https://desk.internal/helpdesk";
    BUG_REPORT_URL = "https://git.desk.internal/platform/deskos/issues";
    ANSI_COLOR = "0;38;2;47;95;117";
    LOGO = "deskos-mark";
    CPE_NAME = "cpe:/o:desk:deskos:26.05";
  };

  system.nixos.extraLSBReleaseArgs = {
```

```
    DISTRIB_DESCRIPTION = "DeskOS 26.05 (Yarara)";
  };

  networking.hostName = lib.mkDefault "deskos";
}
```

Import it from the ISO **and** from `nixosConfigurations.desk-laptop`. After install:

```
cat /etc/os-release
hostnamectl
```

```
NAME=DeskOS
ID=deskos
ID_LIKE=nixos
PRETTY_NAME="DeskOS 26.05 (Yarara)"
VARIANT="DeskOS Workstation"
IMAGE_ID=deskos-installer
```

`ID=deskos` is a lowercase token (os-release). `NAME` is the human string. Do not set `ID=NixOS` and a custom `NAME` — scripts key off `ID`.

Leave `system.nixos.release` / `codeName` alone. Those still say **26.05 / Yarara** because that is the nixpkgs train. A fake `VERSION_ID=1.0` that hides 26.05 is how advisories miss you.

Case 3: Bootloader artwork (GRUB + BIOS splash)

Splash files are PNG (or non-progressive JPEG). Generate placeholders with ImageMagick so the flake builds before the designer ships real art:

```
# modules/desk-iso-brand.nix
{ pkgs, lib, ... }:
let
  splashBios = pkgs.runCommand "deskos-splash-bios.png" {
    nativeBuildInputs = [ pkgs.imagemagick ];
  } ''
    magick -size 800x600 xc:'#1b2838' \
      -gravity center -fill '#7ebae4' -pointsize 48 \
      -annotate 0 'DeskOS' "$out"
  '';
  splashEfi = pkgs.runCommand "deskos-splash-efi.png" {
    nativeBuildInputs = [ pkgs.imagemagick ];
  } ''
    magick -size 1920x1080 xc:'#1b2838' \
      -gravity center -fill '#7ebae4' -pointsize 72 \
      -annotate 0 'DeskOS' "$out"
  '';
```

```
in
{
  isoImage.splashImage = splashBios;
  isoImage.efiSplashImage = splashEfi;
  isoImage.grubTheme = null; # skip nixos-grub2-theme; use colours below
  isoImage.syslinuxTheme = ''
    MENU TITLE DeskOS 26.05
    MENU RESOLUTION 800 600
    MENU COLOR SEL 7;37;40 #FFFFFFFF #FF2F5F75 std
    MENU COLOR UNSEL 37;40 #FF7EBAE4 #00000000 none
  '';

  boot.loader.grub.extraConfig = ''
    set color_normal=light-cyan/black
    set color_highlight=black/light-cyan
    set menu_color_normal=light-cyan/black
    set menu_color_highlight=black/light-cyan
  '';
}
```

Replace the `runCommand` PNGs with `./artwork/splash-bios.png` once the files exist. Keep artwork **in git** (or a FOD), not fetched from a CMS at eval time.

`isoImage.grubTheme` expects a GRUB2 theme directory (`theme.txt` + fonts + background). `pkgs.nixos-grub2-theme` is the NixOS default — set it to `null` or to **your** theme package. Mixing a custom `distroName` with the stock snowflake theme looks like a counterfeit.

`isoImage.contents` drops extra files on the ISO:

```
{
  isoImage.contents = [
    {
      source = ./README-INSTALL.md;
      target = "/README-INSTALL.md";
    }
  ];
}
```

Case 4: Plymouth + live graphical session

For a **graphical** installer, import the Calamares CD instead of minimal:

```
# modules/desk-iso-graphical.nix
{ modulesPath, pkgs, ... }:
{
  imports = [
    (modulesPath + "/installer/cd-dvd/installation-cd-graphical-calamares-plasma6.nix")
```

```
    ./desk-identity.nix
    ./desk-iso-brand.nix
  ];

  isoImage.edition = "plasma6";

  boot.plymouth.enable = true;
  boot.plymouth.theme = "breeze";
  boot.plymouth.logo = ./artwork/logo-48.png; # PNG only; 48×48 matches GDM

  boot.kernelParams = [ "quiet" ];
  boot.consoleLogLevel = 3;
  boot.initrd.verbose = false;

  services.displayManager.sddm.enable = true;
  services.desktopManager.plasma6.enable = true;
}
```

`boot.plymouth.theme = "breeze"` pulls a Plymouth theme whose **osName** is `config.system.nixos.distroNam` — so Case 2's `DeskOS` appears on the splash without a custom theme package. `theme = "bgrt"` keeps the firmware OEM logo; that is often what you want on laptops, **not** on a USB installer.

A live wallpaper (Plasma):

```
{
  environment.etc."deskos/wallpaper.png".source = ./artwork/wallpaper.png;
}
```

Point the Plasma look-and-feel at that path in a small `xdg` autostart if you must; do not vendor all of `plasma-workspace-wallpapers` (it inflates the ISO by hundreds of MiB).

Case 5: Calamares branding (the installer window)

Calamares on the NixOS graphical ISO uses `pkgs.calamares-nixos-extensions`, branding component `nixos`. Override the component with files you own.

Save as `artwork/calamares/branding.desc` (Calamares format; truncated to the strings that matter):

```
# artwork/calamares/branding.desc
---
componentName:  deskos
windowExpanding: fullscreen
windowPlacement: center
sidebar: widget
navigation: widget
strings:
```

```
    productName:         DeskOS
    shortProductName:    DeskOS
    version:             26.05
    shortVersion:        26.05
    versionedName:       DeskOS 26.05
    shortVersionedName:  DeskOS
    bootloaderEntryName: DeskOS
    productUrl:          https://git.desk.internal/platform/deskos
    supportUrl:          https://desk.internal/helpdesk
    knownIssuesUrl:      https://git.desk.internal/platform/deskos/issues
    releaseNotesUrl:     https://git.desk.internal/platform/deskos
    donateUrl:           https://git.desk.internal/platform/deskos
images:
    productIcon:         logo.svg
    productLogo:         logo-white.png
    productWelcome:      logo.svg
slideshow:               show.qml
style:   Raleway
welcomeStyleCalamares:   false
```

Put `logo.svg`, `logo-white.png`, and a one-slide `show.qml` next to that file. Then:

```
# modules/desk-calamares.nix
{ pkgs, ... }:
{
  environment.etc."calamares/branding/deskos".source = ./artwork/calamares;

  environment.etc."calamares/settings.conf".text = ''
    ---
    modules-search: [ local, /run/current-system/sw/lib/calamares/modules ]
    sequence:
      - show: [ welcome, locale, keyboard, partition, users, summary ]
      - exec: [ partition, mount, unpackfs, networkcfg, machineid, locale, keyboard, users,
      - show: [ finished ]
    branding: deskos
    prompt-install: false
    dont-chroot: false
  '';
}
```

`bootloaderEntryName: DeskOS` is what shows up in the firmware boot menu **after** install. If you leave it `NixOS`, the USB said DeskOS and the disk says NixOS — that is the bug users file as "install failed."

Calamares still writes a NixOS `configuration.nix`. After first boot, switch the machine to the **flake** (`nixos-rebuild switch --flake git+ssh://git.desk.internal/platform/deskos#desk-laptop`) so identity + sshd + disk layout are the desk modules, not a one-shot Calamares file. The ISO is a delivery vehicle. The flake is the distro.

The trap

The trap is **a wallpaper and a hostname** with `ID=nixos` still in `/etc/os-release`, plus the official snowflake on the GRUB theme. That is a reskin people will treat as NixOS support. Set `distroId`, URLs, and **your** artwork, keep `ID_LIKE=nixos`, and say "based on NixOS 26.05" in `README-INSTALL.md`.

The other traps:

- `format = "iso"` when you needed `install-iso` / `installation-cd-minimal.nix` — no `nixos-install` on the USB.
- `volumeID` longer than the ISO9660 limit — stage-1 cannot find the CD.
- Copying NixOS logo assets in violation of the NixOS branding guide.
- A password in the ISO module (`users.users.root.password = "desk"`) that ships to every USB in the drawer. SSH keys. Hashed passwords if you must.
- Branding only the ISO, not `nixosConfigurations`. Day-two `nixos-rebuild` restores the snowflake.

The boring rule

- Official installer module (`installation-cd-minimal` or graphical-calamares). 26.05 lock.
- One `desk-identity.nix` imported by ISO **and** hosts. `distroId` lowercase; `ID_LIKE=nixos`.
- Your artwork in git. No snowflake. Splash PNG / GRUB theme / Plymouth logo / Calamares `branding.desc`.
- `isoName` + short `volumeID`. USB bootable (`makeUsbBootable`).
- After install, the flake is source of truth. Calamares is first boot only.
- Do not fork nixpkgs to change a string in `os-release`.

Try this

1. Build Case 1. `iso-info` or `isoinfo -d -i result/iso/*.iso` and read the volume id.
2. Boot the ISO in QEMU; `cat /etc/os-release` — `ID=deskos` and `ID_LIKE=nixos`.
3. Swap Case 3 placeholders for real PNGs; rebuild; confirm GRUB no longer shows `nixos-grub2-theme`.
4. On a graphical ISO, open Calamares and read the window title / `bootloaderEntryName`.
5. Install into a VM, then `nixos-rebuild switch --flake` the same identity module; `hostnamectl` still says DeskOS.

D-Bus Broker Default

D-Bus Broker Default

NixOS 26.05 made **dbus-broker** the default system message bus. Classic `dbus-daemon` still works, but it is no longer the boring path. The boring default is: **leave `services.dbus.implementation = "broker"` (the 26.05 default), treat a bus implementation change as a reboot event (`nixos-rebuild boot` then reboot), and only pin `"dbus"` when you have a measured regression — never to silence a switch-inhibitor warning.**

Stage 2 systemd services (and most of the desktop session) talk over D-Bus. Swapping the bus under a live session closes sockets that may not reconnect. That is why NixOS labels the option as a switch inhibitor.

Mental model

```
apps / services / session

          D-Bus API (same wire protocol)

  message bus implementation
  "broker"  →  dbus-broker       ← NixOS 26.05 default
  "dbus"    →  dbus-daemon       ← opt-out / legacy


    systemd unit alias: dbus.service
```

Concern	Classic `dbus`	`broker` (26.05 default)
Package	`pkgs.dbus`	`pkgs.dbus-broker`
Option value	`"dbus"`	`"broker"`
Goal	Reference daemon	Higher performance and reliability, same D-Bus protocol
Live switch	Unsafe mid-session	Unsafe mid-session — **switch inhibitor**
Safe apply	`nixos-rebuild boot` + reboot	`nixos-rebuild boot` + reboot

On a greenfield 26.05 desk host you rarely set the option at all. Explicit `= "broker"` in snippets below is documentation. Explicit `= "dbus"` is a deliberate opt-out you should be able to justify in one sentence.

The scary message after an upgrade:

```
Checking switch inhibitors...
There are changes to critical components of the system:

dbus-implementation : dbus -> broker

Switching into this system is not recommended.
You probably want to run 'nixos-rebuild boot' and reboot your system instead.
```

That is the system working. Prefer `boot` + reboot. `NIXOS_NO_CHECK=1` forces a live switch that can leave NetworkManager, portals, or the session half-connected to a dead bus.

Worked examples

Case 1: Confirm the bus on a 26.05 desk host

Save as `dbus_probe.nix` and import it from the host flake (optional — defaults already match):

```nix
# dbus_probe.nix
{
  # Explicit for clarity on a 26.05 desk host. Omit on greenfield configs.
  services.dbus.implementation = "broker";
}
```

After boot:

```
nixos-option services.dbus.implementation
systemctl status dbus.service --no-pager | head -n 20
busctl status | head -n 15
```

Representative output on a healthy 26.05 desk host:

```
Value:
"broker"

Default:
"broker"

 dbus.service
  Loaded: loaded (…; enabled; alias of dbus-broker.service)
  Active: active (running)
```

`dbus.service` is an **alias**. Under broker, the real unit is `dbus-broker.service` (and the matching user bus). Apps that `Requires=dbus.service` keep working without knowing which implementation sits behind the name.

Case 2: Upgrade from 25.11 — obey the inhibitor

When the channel/flake input moves to 26.05, the default flips from classic dbus to broker. Evaluation succeeds; activation may refuse a live switch.

```
# Preferred on the desk after a 26.05 bump:
sudo nixos-rebuild boot --flake .#desk-workstation
sudo systemctl reboot
```

Representative inhibitor text if you tried `switch` instead:

```
Checking switch inhibitors...
There are changes to critical components of the system:

dbus-implementation : dbus -> broker

Switching into this system is not recommended.
You probably want to run 'nixos-rebuild boot' and reboot your system instead.
```

Why reboot wins:

1. Old bus stops and closes sockets.
2. New bus starts.
3. Clients that held connections across the restart may never reattach.
4. A reboot starts every consumer against the new bus from scratch.

Save a one-line note in the desk runbook (not in Nix):

```
# DESK-UPGRADE.txt
After bumping nixpkgs to 26.05+: if switch mentions dbus-implementation,
run: nixos-rebuild boot && reboot
Do not export NIXOS_NO_CHECK=1 on a laptop you care about.
```

Case 3: Opt out only with a measured reason

Some rare clients misbehave with broker. Pin classic dbus **after** you can reproduce the failure, not because a forum post looked scary.

Save as `dbus_legacy_optout.nix`:

```
# dbus_legacy_optout.nix
{
  # Temporary incident control - ticket: DESK-1842 (portal disconnect on broker).
  # Revisit after the app ships a fix; do not copy this into every host.
  services.dbus.implementation = "dbus";
}
```

Apply safely:

```
sudo nixos-rebuild boot --flake .#desk-workstation
sudo systemctl reboot
nixos-option services.dbus.implementation
```

```
Value:
"dbus"

Default:
"broker"
```

The drift between Value and Default is your reminder to remove the pin later.

Case 4: Services that depend on the bus still say `dbus.service`

Desk units should keep depending on the **alias**, not on `dbus-broker.service` by name. That keeps the flake portable if you temporarily opt out.

Save as `desk_bus_consumer.nix`:

```
# desk_bus_consumer.nix
{ pkgs, ... }:

{
  services.dbus.implementation = "broker";

  systemd.services.desk-bus-ping = {
    description = "Desk: oneshot that needs a system bus";
    wantedBy = [ "multi-user.target" ];
    after = [ "dbus.service" ];
    wants = [ "dbus.service" ];
    serviceConfig = {
      Type = "oneshot";
      RemainAfterExit = true;
      # Store paths only.
      ExecStart = "${pkgs.systemd}/bin/busctl --system list";
    };
  };
}
```

```
sudo nixos-rebuild boot --flake .#desk-workstation && sudo systemctl reboot
# after reboot:
systemctl status desk-bus-ping --no-pager
busctl --system list | head
```

Output (shape):

```
  desk-bus-ping.service - Desk: oneshot that needs a system bus
      Active: active (exited)
…
org.freedesktop.DBus
org.freedesktop.systemd1
…
```

Hard-coding `after = [ "dbus-broker.service" ];` breaks the day you set `implementation = "dbus"` on a lab host.

Case 5: Dry-run the inhibitor without guessing

Before you touch a remote desk host, inspect the option merge and prefer `boot` whenever the implementation changes between generations.

```
nixos-option services.dbus.implementation
sudo nixos-rebuild dry-activate --flake .#desk-workstation
```

If dry-activate (or a refused switch) reports a dbus implementation change, schedule a reboot window. For unattended desks, put the upgrade in a two-step job: build + `boot`, then a controlled reboot — never `switch` across a bus swap over SSH without a console.

Lab-only force (do not normalize this):

```
# Lab VM with serial console only - never on the family laptop.
NIXOS_NO_CHECK=1 sudo nixos-rebuild switch --flake .#lab-vm
```

The trap

Trap A — force the live switch with `NIXOS_NO_CHECK=1` to "save time." You can end up with a graphical session, NetworkManager, or portals still holding sockets to the old bus. The machine looks up; half the desktop is not. Reboot anyway — you paid the risk and still need the reboot.

Trap B — pin `services.dbus.implementation = "dbus"` on every host to avoid the warning. You freeze the desk on the legacy daemon and re-learn the warning on the next policy cleanup. Leave the 26.05 default; handle the one-time upgrade with `boot` + reboot.

Trap C — depend on `dbus-broker.service` by concrete name in custom units. The portable dependency is `dbus.service`. Broker installs that alias; classic dbus provides the same unit name.

```
# Anti-pattern - cargo-cult opt-out
{
  services.dbus.implementation = "dbus";  # "so switch never complains"
}
```

```
# Boring fix - accept the default; change process, not the bus
{
  # services.dbus.implementation omitted → "broker" on 26.05
}
# Apply bus-affecting upgrades with:
#   sudo nixos-rebuild boot --flake .#desk-workstation && sudo systemctl reboot
```

The boring rule

- On NixOS 26.05+, keep dbus-broker (`"broker"`). Do not set the option unless you are opting out with a ticket.
- Implementation changes are reboot events: `nixos-rebuild boot`, then reboot. Do not normalize `NIXOS_NO_CHECK=1`.
- Depend on `dbus.service`, not `dbus-broker.service`, in custom units.
- Confirm with `nixos-option services.dbus.implementation` and `systemctl status dbus.service`.
- Opt out to `"dbus"` only for a reproduced client bug; remove the pin when the bug is gone.
- Treat the inhibitor message as documentation, not an error to silence.

Try this

1. On a disposable 26.05 VM, run `nixos-option services.dbus.implementation` and confirm `"broker"`. Set `"dbus"`, `nixos-rebuild boot`, reboot, confirm Value/Default drift, then remove the pin and reboot back.
2. From a 25.11 generation (or a VM snapshot), bump the flake input to 26.05 and attempt `nixos-rebuild switch`. Read the inhibitor text. Re-run with `boot` and reboot. Compare how the session feels vs forcing `NIXOS_NO_CHECK=1` once in the lab.
3. Add Case 4's `desk-bus-ping` unit. Flip implementation to `"dbus"` and back without changing the unit's `after = [ "dbus.service" ];`. Confirm it still orders correctly.
4. `systemctl show dbus.service -p Id -p Names -p FragmentPath` under broker and under classic dbus. Note the alias vs concrete unit names.
5. Grep the desk flake for `NIXOS_NO_CHECK`, `implementation = "dbus"`, and hard-coded `dbus-broker.service` dependencies. Open tickets for each hit that lacks a justification comment.

Home Manager

Home Manager and the Module System

Home Manager and the Module System

NixOS configures the machine (kernel, systemd, users, `/etc`). Home Manager configures **one user** (dotfiles, user packages, user services). The boring default is: **`home.packages` for binaries, `programs.<name>` for software that has a module, and `home.stateVersion` left at the birth release**.

Mental model

Home Manager is the NixOS module system aimed at `$HOME`:

Field	Effect
`home.username` / `home.homeDirectory`	Who this config is for
`home.stateVersion`	Birth release. Do not bump casually.
`home.packages`	On the user `PATH` (`~/.nix-profile/bin`)
`programs.<name>.enable`	Typed module: writes dotfiles and maybe a user systemd unit
`home.file` / `xdg.configFile`	Raw files into `$HOME` / `~/.config`

Two ways to apply it:

Mode	Command	When
Standalone	`home-manager switch --flake .#deskadmin`	Nix-on-Linux, macOS, or a user who must not touch NixOS
NixOS module	`nixos-rebuild switch`	The desk workstation is NixOS (next chapters in this part)

Standalone is how you start on a foreign distro. The NixOS module is how you stay once the machine is yours.

```
home.nix
      home.packages        →  ~/.nix-profile
      programs.git         →  ~/.config/git/config
      xdg.configFile       →  ~/.config/…


         home-manager switch
```

Worked examples

Case 1: A standalone `home.nix`

Save as `home.nix`:

```
# home.nix
{ pkgs, ... }:

{
  home.username = "deskadmin";
  home.homeDirectory = "/home/deskadmin";
  home.stateVersion = "26.05";

  home.packages = with pkgs; [
    bat
    fzf
    jq
    ripgrep
  ];

  programs.bash.enable = true;
  programs.home-manager.enable = true;

  # Existing ~/.config/git/config is renamed, not clobbered.
  home.backupFileExtension = "hmbak";
}
```

On a machine with the `home-manager` CLI:

```
home-manager switch -f home.nix
```

Output:

```
Starting Home Manager activation
Activating checkFilesChanged
Activating installPackages
…
Home Manager switch complete!
```

```
which jq
jq --version
```

`jq` resolves under `~/.nix-profile/bin` or `~/.local/state/nix/profiles/home-manager/…`.

Case 2: A flake with `homeConfigurations`

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk user environment";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    home-manager = {
      url = "github:nix-community/home-manager/release-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, home-manager }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      homeConfigurations.deskadmin = home-manager.lib.homeManagerConfiguration {
        inherit pkgs;
        modules = [ ./home.nix ];
      };
    };
}
```

`inputs.nixpkgs.follows` prevents Home Manager from pulling a second nixpkgs.

```
home-manager switch --flake .#deskadmin
```

Pin both inputs in `flake.lock`. The same lock on a second laptop is the same `jq`.

Case 3: User packages are not system services

`home.packages = [ pkgs.nginx ];` puts `nginx` on **your** PATH. It does not open port 80 and it does not start a daemon. Daemons belong in NixOS `services.*` (or a user systemd module you wrote on purpose).

```
# wrong mental model
home.packages = [ pkgs.openssh ];  # a binary, not sshd
```

```
# NixOS, not Home Manager
services.openssh.enable = true;
```

Case 4: Generation rollback for `$HOME`

```
home-manager generations
home-manager switch --rollback
```

Output:

```
2026-09-09 10:11 : id 42 -> /nix/store/…-home-manager-generation
2026-09-08 18:02 : id 41 -> /nix/store/…-home-manager-generation
Switching to generation 41
```

Broken `programs.neovim`? Roll back. You do not `rm ~/.config/nvim`.

Case 5: `stateVersion` is a birth certificate

```
home.stateVersion = "26.05";
```

This is **not** "track 26.05." It tells Home Manager which defaults your state dirs were created with. When you upgrade nixpkgs to a newer release, you keep `stateVersion` until a release note says a specific option needs a bump.

The trap

The trap is **managing the same program in both NixOS and Home Manager** (`environment.systemPackages = [ git ]` **and** `programs.git.enable = true` with conflicting `userName`). Two `git` binaries, two configs, a week of "why is my email wrong."

Pick a layer: git config for humans → Home Manager. git as a system tool for root scripts → NixOS. Not both with different emails.

The other trap is running standalone Home Manager against a different nixpkgs than NixOS on the same box. `follows` in the flake, or the NixOS module with `useGlobalPkgs` (later chapter).

A third: `home.file.".gitconfig".text = …` **and** `programs.git.enable = true`. Two writers, last activation wins. Use the `programs.git` module; `home.file` is for files that have no module.

The boring rule

- Home Manager owns `$HOME`. NixOS owns the machine.
- `home.packages` for extra CLIs. `programs.<name>` when a module exists. `home.file` only when no module exists.
- `home.backupFileExtension` so the first switch does not refuse a pre-existing dotfile.
- Flake + `follows = "nixpkgs"` so there is one nixpkgs.
- `home.stateVersion` stays at the birth release.
- Roll back with `home-manager switch --rollback`, not by deleting dotfiles.

Try this

1. Add `pkgs.hello` to `home.packages`, switch, run `hello`, remove it, switch again, and confirm `which hello` fails.
2. `readlink -f $(which jq)` and confirm the path is under `/nix/store`.
3. Run `home-manager generations` and identify the current id.
4. In the flake, omit `inputs.nixpkgs.follows` on purpose, `nix flake metadata`, and notice a second nixpkgs. Put `follows` back.

Managing Dotfiles Declaratively

Managing Dotfiles Declaratively

Git-bare + `stow` leaves stale symlinks and a README of "also copy this file." The boring default is: **`xdg.configFile` and `home.file` so every live config is a symlink into `/nix/store`, generated from this Home Manager generation**.

Mental model

Option	Live path
`xdg.configFile."desk/settings.json"`	`~/.config/desk/settings.json`
`home.file.".desk.rc"`	`~/.desk.rc`
`xdg.dataFile."desk/banner.txt"`	`~/.local/share/desk/banner.txt`

Each value can be `.text` (inline string), `.source` (a file in git), or `.template` (less common). Home Manager creates the parent directories and a generation-specific symlink.

The file in `$HOME` is **read-only** when it points at the store. That is the feature: if you need to change it, you change git and switch. An editor that "helpfully" writes in place will get `Permission denied` or silently copy-break the symlink depending on the editor.

`xdg.configFile."desk/settings.json".force = true;` replaces a leftover regular file. Prefer `home.backupFileExtension` (previous chapter) so the first switch copies `*.hmbak` instead of aborting. `onChange` runs a command after the file is replaced (`systemctl --user restart desk-sync.service`) — keep it short; it is not a substitute for a real user unit.

```
git: desk/settings.json
           home.file / xdg.configFile

/nix/store/…-home-manager-files/.config/desk/settings.json

           ~/.config/desk/settings.json  (symlink)
```

Worked examples

Case 1: Inline JSON from Nix

Save as `dotfiles.nix`:

```
# dotfiles.nix
{
  xdg.configFile."desk/settings.json".text = builtins.toJSON {
    theme = "dark";
    autoSync = true;
    window = "A";
  };
}
```

Import this from `home.nix`. After `home-manager switch`:

```
readlink -f ~/.config/desk/settings.json
cat ~/.config/desk/settings.json
```

Output:

```
/nix/store/…-settings.json
{"autoSync":true,"theme":"dark","window":"A"}
```

`builtins.toJSON` keeps commas honest. Do not hand-write JSON in a Nix multiline string if you can avoid it.

Case 2: Source a file that lives in git

Save as `gitconfig-extra.txt` next to the module:

```
# gitconfig-extra.txt
[alias]
    st = status
    co = checkout
```

Save as `dotfiles-source.nix`:

```
# dotfiles-source.nix
{
  xdg.configFile."git/ignore".text = ''
    result
    .direnv/
    *~
  '';

  home.file.".desk/aliases".source = ./gitconfig-extra.txt;
}
```

```
home-manager switch
cat ~/.desk/aliases
```

The source file is copied into the store at build time. Editing `~/.desk/aliases` does not change git. Editing `./gitconfig-extra.txt` and switching does.

Case 3: On-change hook

Save as `onchange.nix`:

```
# onchange.nix
{
  xdg.configFile."desk/reload-flag".text = "reload-v1";
  home.activation.deskPing = ''
    echo "desk configs refreshed at $(date -u +%H:%M:%S)" >> "$HOME/.desk/activation.log"
  '';
}
```

Prefer dedicated modules (`programs.git`, `programs.ssh`) over `home.activation` scripts. Activation runs as your user on every switch; keep it idempotent.

Case 4: Mutable file when a program insists on writing

Some tools rewrite their config on every start (a desktop app dumping window geometry). Forcing a store symlink makes the app crash or clone the file.

```
# mutable.nix
{
  xdg.configFile."desk/geometry.json" = {
    text = builtins.toJSON { x = 0; y = 0; };
    force = true;
  };
}
```

`force = true` replaces a **user-created** file with the symlink. It does not make the store writable. If the app must write, **do not** manage that file. Manage a `config.example` and leave the live file alone.

Case 5: Inspect what this generation linked

```
home-manager files | head
ls -l ~/.config/desk
```

Every managed path should be a symlink into `/nix/store`. A regular file at a path you thought you managed means the app (or you) replaced the symlink.

The trap

The trap is **`chmod u+w ~/.config/desk/settings.json` and editing in place**. The next `home-manager switch` either overwrites you or refuses to clobber. Your change was never in git.

The other trap is stowing **and** Home Manager on the same path. Two tools fight. Pick Home Manager; delete the stow tree.

The boring rule

- Config that should be identical on every desk machine lives in git and is linked via `xdg.configFile` / `home.file`.
- Prefer `.text` + `toJSON` / real modules over huge copied blobs.
- Store symlinks are read-only. Edit the source, then switch.
- If an app writes the file, do not manage that path.
- Do not mix stow, chezmoi, and Home Manager on one directory.

Try this

1. Add `xdg.configFile."desk/motd".text = "window A";`, switch, `cat ~/.config/desk/motd`.
2. `ls -l ~/.config/desk/motd` — confirm a symlink. Try `echo x >> ~/.config/desk/motd` and record the error.
3. Change the motd text, switch, confirm the store path hash on `readlink -f` moved.
4. Create a real file at `~/.config/desk/motd` after `rm` of the symlink, switch without `force`, and read Home Manager's conflict message. Then restore management with `force = true` **once**, and stop using force as a habit.

Shell and Editor Configuration

Shell and Editor Configuration

Copying `.zshrc` snippets between laptops is how aliases drift. The boring default is: **typed `programs.*` modules for the shell, prompt, git, and editor, with plugins fetched by Nix instead of a plugin manager that writes into `$HOME`.**

Mental model

Home Manager modules know the file format. `programs.git` writes `~/.config/git/config`. `programs.starship` writes `~/.config/starship.toml`. You set Nix attributes; you do not maintain a parallel dotfile unless the module is missing.

Module	Replaces
`programs.bash` / `zsh` / `fish`	Hand-written rc files for aliases, history, completions
`programs.starship`	Prompt theme copied from a gist
`programs.git`	`git config --global`
`programs.neovim` / `programs.vim`	Plugin managers (`vim-plug` curling GitHub on first launch)

Enable the shell you actually log into. Enabling `programs.zsh` does not change `/etc/passwd`. The **login shell** is a NixOS `users.users.deskadmin.shell = pkgs.zsh;` (and `programs.zsh.enable = true;` on NixOS for `/etc/zshrc`). On a foreign distro, `chsh` once after Home Manager installed zsh in the profile.

`programs.neovim.plugins` takes nixpkgs vimPlugins, not `Plug 'foo'` that curls GitHub on first launch. A `init.lua` via `xdg.configFile."nvim/init.lua"` is fine when the module's `extraLuaConfig` is too small — still no packer sync at runtime.

Worked examples

Case 1: Git as a module, not a ritual

Save as `shell_editor.nix`:

```
# shell_editor.nix
{ pkgs, ... }:

{
```

```
  programs.git = {
    enable = true;
    userName = "Desk Engineer";
    userEmail = "desk@corp.internal";
    extraConfig = {
      init.defaultBranch = "main";
      pull.rebase = true;
      rerere.enabled = true;
    };
    aliases = {
      st = "status";
      co = "checkout";
    };
    ignores = [ "result" ".direnv/" ];
  };
}
```

```
home-manager switch
git config --global --list | head
```

Output:

```
user.name=Desk Engineer
user.email=desk@corp.internal
init.defaultbranch=main
alias.st=status
```

`git config --global user.email` in a terminal will write a **different** file or fight the symlink. Stop doing that.

Case 2: Starship prompt

```
# starship.nix
{
  programs.starship = {
    enable = true;
    enableBashIntegration = true;
    settings = {
      add_newline = false;
      format = "$directory$git_branch$character";
      character.success_symbol = "[>](bold green)";
    };
  };

  programs.bash.enable = true;
```

```
}
```

Starship's native format is TOML. Home Manager accepts a Nix attrset and renders TOML. You do not keep a second `starship.toml` in git unless you source it wholesale with `.source`.

Case 3: Bash aliases and history

```
# bash.nix
{
  programs.bash = {
    enable = true;
    historyControl = [ "ignoredups" "erasedups" ];
    historySize = 10000;
    shellAliases = {
      ll = "ls -l";
      gs = "git status";
      ".." = "cd ..";
    };
    bashrcExtra = ''
      # rare exceptions that have no module option
      export DESK_WINDOW=A
    '';
  };
}
```

`shellAliases` is the boring place for aliases. `bashrcExtra` is an escape hatch; if it grows past twenty lines, look for a module.

Case 4: Neovim with plugins from nixpkgs

```
# nvim.nix
{ pkgs, ... }:

{
  programs.neovim = {
    enable = true;
    viAlias = true;
    vimAlias = true;
    extraConfig = ''
      set number
      set relativenumber
    '';
    plugins = with pkgs.vimPlugins; [
      vim-nix
      vim-commentary
```

```
    ];
  };
}
```

Plugins are store paths. There is no `~/.local/share/nvim/site/pack` filled by a job on first open. CI can `nvim --headless +q` without a network.

Case 5: Login shell versus module

On NixOS:

```
# in configuration.nix
users.users.deskadmin.shell = pkgs.bashInteractive;
```

Home Manager's `programs.bash.enable = true` writes `~/.bashrc`. It does not change the passwd shell. If you enable `programs.zsh` but leave `shell = pkgs.bashInteractive`, you will wonder why zsh settings never run.

The trap

The trap is **Oh My Zsh / a plugin manager plus Home Manager**. OMZ clones repos into `$HOME` on the next login and overwrites what Home Manager just linked. Pick one. The boring pick is Home Manager + nixpkgs plugins.

The other trap is putting secrets in `extraConfig` (`smtp.password`, GitHub tokens). Those land in the world-readable store. Use sops-nix / agenix (secrets part) and point the module at the decrypted path.

The boring rule

- If a `programs.<name>` module exists, use it.
- Git identity and aliases live in `programs.git`, not in a one-off `git config`.
- Plugins come from `pkgs.vimPlugins` / `pkgs.zshPlugins`, not from a curl-on-login manager.
- Set the login shell in NixOS (or `chsh` once on a foreign distro).
- No secrets in module extraConfig.

Try this

1. Add `programs.git.aliases.lg = "log --oneline --graph -20";`, switch, run `git lg`.
2. `readlink -f ~/.config/git/config` and confirm a store path.
3. Enable starship, open a new bash, and confirm `echo $STARSHIP_SESSION_KEY` is set (or that the prompt changed). Disable and switch to compare.

4. List `pkgs.vimPlugins` in a `nix repl` (`:l <nixpkgs>` then `pkgs.vimPlugins.vim-nix`) to see that the plugin is an ordinary derivation.

Application Settings and Preferences

Application Settings and Preferences

CLIs are not the only thing in `$HOME`. Terminals, editors, and browsers all grow JSON/TOML/dconf trees. The boring default is: **use a Home Manager module when one exists; generate the native file from Nix; do not manage files the app rewrites on every launch**.

Mental model

Home Manager modules compile Nix into the format the app already understands:

Module	Writes
`programs.alacritty`	`~/.config/alacritty/alacritty.toml`
`programs.kitty`	`~/.config/kitty/kitty.conf`
`programs.vscode`	`settings.json` + extensions from nixpkgs
`programs.ssh`	`~/.ssh/config`
`programs.tmux`	`~/.config/tmux/tmux.conf`

If there is no module, `xdg.configFile."app/config".text` is still better than a README. If the app treats the config as a database (window geometry, crash dumps, "recent files"), leave it unmanaged. Home Manager 26.05 will **fail the switch** when it tries to overwrite a file the app rewrote, unless you set `home-manager.backupFileExtension` (NixOS module) or stop managing that path.

Skip: browser profiles, `~/.config/Code/User/globalStorage`, `~/.local/share/*/state`, anything SQLite. Manage: the TOML/JSON the app documents as configuration.

Worked examples

Case 1: Alacritty

Save as `apps.nix`:

```nix
# apps.nix
{ pkgs, ... }:

{
  programs.alacritty = {
    enable = true;
```

```
    settings = {
      window.opacity = 0.95;
      font.size = 12.0;
      font.normal.family = "JetBrains Mono";
      colors.primary.background = "#1e1e1e";
    };
  };
}
```

```
home-manager switch
test -f ~/.config/alacritty/alacritty.toml && echo ok
```

Output:

```
ok
```

Open Alacritty. Opacity and font come from this generation. A distro-packaged Alacritty that ignores XDG is a different binary — install Alacritty via the same Home Manager module (`programs.alacritty.enable` already installs the package).

Case 2: SSH client config

```
# ssh.nix
{
  programs.ssh = {
    enable = true;
    # Home Manager 26.05: RFC 42 programs.ssh.settings is the new form.
    # matchBlocks still works; it is deprecated and migrated automatically.
    extraConfig = ''
      Host *
        IdentityFile ~/.ssh/id_ed25519
        IdentitiesOnly yes
    '';
    matchBlocks = {
      "desk-bastion" = {
        hostname = "bastion.desk.internal";
        user = "deskadmin";
        port = 22;
      };
      "desk-*" = {
        proxyJump = "desk-bastion";
        user = "deskadmin";
      };
    };
  };
```

```
}
```

Host keys and **private** keys are not in this file. `IdentityFile` **points** at `~/.ssh/id_ed25519`, which you create with `ssh-keygen` or decrypt with sops. Putting a private key in `.text` copies it into `/nix/store`.

When you migrate, `programs.ssh.settings` is the 26.05 knob (same ideas: Host, HostName, User). Do not invent a second `~/.ssh/config` via `home.file` while the module is enabled — two writers, one file.

Case 3: tmux

```
# tmux.nix
{ pkgs, ... }:

{
  programs.tmux = {
    enable = true;
    clock24 = true;
    keyMode = "vi";
    terminal = "tmux-256color";
    extraConfig = ''
      bind r source-file ~/.config/tmux/tmux.conf
    '';
    plugins = with pkgs.tmuxPlugins; [ sensible yank ];
  };
}
```

Plugins are store paths, same story as neovim.

Case 4: VS Code extensions from nixpkgs

```
# vscode.nix
{ pkgs, ... }:

{
  programs.vscode = {
    enable = true;
    # 26.05: extensions and userSettings live on the profile.
    # Top-level userSettings is obsolete (renamed to profiles.default.*).
    profiles.default = {
      extensions = with pkgs.vscode-extensions; [
        jnoortheen.nix-ide
        tamasfe.even-better-toml
      ];
```

```
      userSettings = {
        "editor.tabSize" = 2;
        "files.insertFinalNewline" = true;
      };
    };
  };
}
```

Marketplace extensions that are **not** in nixpkgs will not appear here. That is inconvenient and correct: an unsigned VSIX from the internet is not a pinned input. Use `pkgs.vscode-utils.extensionFromVscodeMarketplace` only when you pin `sha256`.

VS Code **forks** on Home Manager 26.05 have their own modules (`programs.vscodium`, `programs.cursor`, …). Do not set `programs.vscode.package = pkgs.vscodium` — that `pname` knob was removed.

Do **not** manage `~/.config/Code/User/globalStorage` or `workspaceStorage`. Those directories are rewritten on every launch. If `home-manager switch` errors `… exists but is not a symlink`, either `backupFileExtension = "bak"` once, or remove the path from Home Manager. Do not `force = true` on a database the editor holds open.

Case 5: Fonts the terminal named

```
# fonts.nix
{ pkgs, ... }:

{
  fonts.fontconfig.enable = true;
  home.packages = with pkgs; [
    jetbrains-mono
    nerd-fonts.jetbrains-mono
  ];
}
```

`programs.alacritty.settings.font.normal.family = "JetBrains Mono";` does nothing if the font is not installed. Put the font in `home.packages` (or NixOS `fonts.packages`).

The trap

The trap is **managing `~/.config/Code/User/globalStorage`** or a browser profile. Those directories are mutable state. Home Manager will either clobber your session or fail the switch.

The other trap is committing `programs.ssh.extraConfig` with `IdentityFile /nix/store/…-id_ed25519`. That store path is world-readable. Keys live outside the store, or in a sops-decrypted tmpfs.

A third: `programs.vscode.userSettings` on 26.05 (obsolete — use `profiles.default`). A fourth: `force = true` on a file the app rewrites so every `switch` fights the running editor.

The boring rule

- Modules first, `xdg.configFile` second, unmanaged mutable state last.
- Terminal + font + editor settings are code. Window geometry, `globalStorage`, browser profiles are not.
- SSH **config** is code (`settings` / migrated `matchBlocks`). SSH **keys** are not.
- VS Code: `profiles.default` on 26.05; extensions from nixpkgs (or a pinned marketplace fetch). Forks get their own module.
- If `home-manager switch` fights the app, stop managing that path. `backupFileExtension` is a one-shot, not a lifestyle.

Try this

1. Set Alacritty `font.size` to `14.0`, switch, open a new terminal, then set it back.
2. Add an SSH `matchBlocks` entry for a lab host and `ssh -G desk-bastion | grep hostname`.
3. `readlink -f ~/.config/alacritty/alacritty.toml` — store path.
4. Intentionally add a private key as `home.file.".ssh/id_ed25519".text = "-----BEGIN …";` in a **throwaway** VM, `nix-store -q --hash` that path, then delete it and GC. That hash is why we never do this.
5. Launch Alacritty, change opacity in the GUI if it has one, `switch` again. If HM fights, that file is not yours — drop it from the module.

Combining NixOS and Home Manager

Combining NixOS and Home Manager

Two rebuild commands (`nixos-rebuild` and `home-manager switch`) means two nixpkgs, two lock files, and a rollback that restores the kernel but not `~/.bashrc`. The boring default on a NixOS workstation is: **Home Manager as a NixOS module, one flake, one `nixos-rebuild switch`**.

Mental model

```
flake.nix
  nixosConfigurations.desk-vm
          NixOS modules (boot, users, services)
          home-manager.nixosModules.home-manager
                 home-manager.users.deskadmin = { … };
```

`nixos-rebuild switch --flake .#desk-vm` then:

1. Builds the system closure.
2. Builds the user's Home Manager generation.
3. Activates both.
4. Records **one** system generation that includes the home generation.

`--rollback` restores both.

Option	Why
`home-manager.useGlobalPkgs = true;`	One nixpkgs: the system's. No second eval.
`home-manager.useUserPackages = true;`	User packages land in the NixOS user profile, not a disconnected HM profile
`home-manager.users.<name>`	Must match `users.users.<name>`

Standalone Home Manager remains correct on a foreign distro (next chapter). On NixOS, the module wins.

Worked examples

Case 1: Minimal unified module

Save as `unified.nix`:

```
# unified.nix
{ config, pkgs, ... }:

{
  home-manager.useGlobalPkgs = true;
  home-manager.useUserPackages = true;

  home-manager.users.deskadmin = { pkgs, ... }: {
    home.stateVersion = "26.05";
    programs.git = {
      enable = true;
      userName = "Desk Engineer";
      userEmail = "desk@corp.internal";
    };
    home.packages = [ pkgs.jq ];
  };
}
```

The user `deskadmin` must already exist in NixOS (`users.users.deskadmin.isNormalUser = true;`). Home Manager will not create a system user.

Case 2: Flake wiring

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk workstation";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    home-manager = {
      url = "github:nix-community/home-manager/release-26.05";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, home-manager }: {
    nixosConfigurations.desk-vm = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        ./configuration.nix
        home-manager.nixosModules.home-manager
        ./unified.nix
      ];
    };
```

```
  };
}
```

```
sudo nixos-rebuild switch --flake .#desk-vm
```

Output:

```
building the system configuration...
activating the configuration...
Starting Home Manager activation
…
```

One command. `jq` is on the user's PATH after login.

Case 3: Split files per user

Save as `home/deskadmin.nix`:

```
# home/deskadmin.nix
{ pkgs, ... }:

{
  home.stateVersion = "26.05";
  programs.bash.enable = true;
  home.packages = with pkgs; [ ripgrep jq ];
}
```

In `unified.nix`:

```
# unified.nix
{
  home-manager.useGlobalPkgs = true;
  home-manager.useUserPackages = true;
  home-manager.users.deskadmin = import ./home/deskadmin.nix;
}
```

Do not invent a second flake output `homeConfigurations.deskadmin` **on the same NixOS host** unless you have a reason to apply it separately (you probably do not).

Shared user modules (git, editor) that every human should get:

```
# unified.nix fragment
{
  home-manager.sharedModules = [ ./home/common.nix ];
  home-manager.extraSpecialArgs = { deskDomain = "desk.internal"; };
}
```

`extraSpecialArgs` is how a home module receives flake inputs (`inputs.nixpkgs` is already `pkgs` when `useGlobalPkgs` is on). Do not import the system `config` into home unless you must — it is a cycle magnet.

Case 4: Backup files on clobber

```
{
  home-manager.backupFileExtension = "bak";
}
```

If a real file is sitting where Home Manager wants a symlink, the switch copies it to `*.bak` instead of aborting. Useful once during migration. If you see new `.bak` files every week, something is still writing in place.

Case 5: Rollback includes `$HOME`

```
sudo nixos-rebuild switch --rollback
```

The previous system generation's Home Manager generation comes back. `~/.config/git/config` points at the old store path. You do not run `home-manager switch --rollback` in addition.

The trap

The trap is **keeping a standalone `home-manager switch --flake .#deskadmin` cron on a NixOS box** that also imports the module. Two tools write `~/.bashrc`. The last one to run wins. Disable the standalone CLI in muscle memory once the module is on.

The other trap is `useGlobalPkgs = false` "so I can pin a newer git." You now evaluate nixpkgs twice and can mix library ABIs in one user session. If you need a newer git, overlay it on the **system** nixpkgs.

The boring rule

- On NixOS: Home Manager as a NixOS module. One flake. One rebuild.
- `useGlobalPkgs = true; useUserPackages = true;`.
- `home-manager.users.<name>` matches `users.users.<name>`.
- `follows = "nixpkgs"` on the home-manager input. `sharedModules` for team defaults.
- `backupFileExtension` at the **home-manager.** NixOS option (not only inside `home.nix`) so clobber policy is one place.
- Rollback is `nixos-rebuild switch --rollback`. Stop running two CLIs.

Try this

1. Wire Case 2 on a lab VM, add `pkgs.hello` to the user's `home.packages`, switch, run `hello` as `deskadmin`.
2. `ls /nix/var/nix/profiles/system` and find the home-manager generation referenced from the current system.
3. Change git `userEmail`, switch, then `--rollback` and confirm the old email is back.
4. Grep your flake for `homeConfigurations`. If it exists **and** `nixosConfigurations` applies the same user, delete the duplicate output.

CI/CD with Nix

Setting Up Binary Caches

Setting Up Binary Caches

Compiling llvm on every laptop is not a culture. The boring default is: **a signed substituter (cache.nixos.org plus one team cache), public keys in `nix.conf`, and CI that uploads what it built**.

Mental model

When Nix needs `/nix/store/<hash>-pkg` it:

1. Looks in the local store.
2. Queries each **substituter** for a `.narinfo` signed with a **trusted public key**.
3. Downloads the NAR if the signature matches.
4. Builds from the `.drv` only if nobody had the bytes.

Piece	Role
Substituter URL	`https://cache.nixos.org`, `https://desk.cachix.org`, `http://attic.desk.internal`
Trusted public key	`cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQF`
Signing private key	Lives on CI / the cache host. Never in git.

Cachix is the hosted boring choice. Attic or Harmonia is the self-hosted one. The NixOS module is the same shape: extra substituters + extra keys.

```
laptop  --query-->  desk-cache  --miss-->  cache.nixos.org  --miss-->  build
CI      --build-->  sign  --put-->  desk-cache
```

A substituter **without** its public key is silently skipped (or refused). You compile the world and blame the network.

`nix.settings.substituters` **replaces** the default list if you assign it — keep `https://cache.nixos.org` in the list. `extra-substituters` **appends**. Same for keys: `trusted-public-keys` vs `extra-trusted-public-keys`. `require-sigs` stays true (the default). `nixConfig.extra-substituters` inside `flake.nix` only applies when the user/daemon `accept-flake-config` (or a trusted user); it is not a substitute for `/etc/nix/nix.conf`. Never put a signing secret in `nixConfig`.

Worked examples

Case 1: Consume a team cache on NixOS

Save as `cache_config.nix`:

```
# cache_config.nix
{
  nix.settings = {
    substituters = [
      "https://cache.nixos.org"
      "https://desk-cache.cachix.org"
    ];
    trusted-public-keys = [
      "cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY="
      "desk-cache.cachix.org-1:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa="
    ];
    require-sigs = true;
  };
}
```

Replace the `desk-cache` key with the real key Cachix showed when you created the cache. A truncated key in this book is **not** trusted; Nix will ignore that substituter.

```
sudo nixos-rebuild switch
nix show-config | grep -E 'substituters|trusted-public-keys'
```

On a multi-user machine the **daemon** reads `/etc/nix/nix.conf`. `~/.config/nix/nix.conf` is ignored for daemon builds.

Untrusted users cannot add substituters. If a laptop must use the team cache without being root:

```
# extraTrusted
{
  nix.settings.trusted-users = [ "root" "deskadmin" ];
}
```

Prefer listing the cache in the **system** `nix.settings` (Case 1) so every uid gets it. `trusted-users` is also who may import unsigned paths — keep it short.

Case 2: Push from a workstation (once)

```
nix build .#desk-api
cachix watch-exec desk-cache -- nix build .#desk-api
```

or:

```
nix build .#desk-api
cachix push desk-cache ./result
```

`CACHIX_AUTH_TOKEN` is an environment variable from the Cachix dashboard. It is a secret. It is not a flake input. Laptops pull; CI pushes. Do not require every laptop to upload.

Case 3: Verify a hit versus a miss

```
nix build .#desk-api --print-build-logs 2>&1 | tee /tmp/desk-build.log
grep -E 'copying path|building' /tmp/desk-build.log | head
```

`copying path '…' from 'https://desk-cache.cachix.org'` is a hit. `building '…drv'` is a miss. A team that still builds llvm locally has a key mismatch or is pushing different hashes (different nixpkgs lock).

```
nix build nixpkgs#hello --max-jobs 0
```

`--max-jobs 0` forbids a local compile. If hello errors, the substituter list is empty or unsigned.

```
nix show-config | grep -E 'substituters|trusted-public-keys'
curl -fsS https://desk-cache.cachix.org/nix-cache-info | head
```

`StoreDir: /nix/store` in `nix-cache-info` is a live cache. HTTP 404 here is why every laptop rebuilds llvm.

Case 4: Serve a tiny cache with `nix copy`

Without Cachix, an S3 bucket or a directory works:

```
nix copy --to file://$PWD/desk-nar .#desk-api
nix copy --from file://$PWD/desk-nar --to ssh://deskadmin@desk-vm .#desk-api
```

`file://` is a lab. Production is HTTP + signatures (`nix-serve`, Harmonia, Attic). Unsigned `file://` on a USB stick is fine for air-gap day; it is not a fleet cache.

Case 5: Generate a signing key for a self-hosted cache

```
nix key generate-secret --key-name desk-cache > /run/secrets/desk-cache.secret
nix key convert-secret-to-public < /run/secrets/desk-cache.secret
```

Put the **public** half in `trusted-public-keys`. Put the secret on the builder only (sops). `nix copy --to 's3://desk-cache?secret-key=/run/secrets/desk-cache.secret'` signs on upload.

The trap

The trap is **adding a substituter without the public key**. Nix will skip it. You will compile the world and blame the network.

The other trap is putting the **private** signing key in `flake.nix` or in a GitHub Actions log (`echo $CACHIX_AUTH_TOKEN`). Rotate immediately.

A third trap: `substituters` in `~/.config/nix/nix.conf` on a multi-user machine. The daemon reads `/etc/nix/nix.conf`.

A fourth: assigning `substituters = [ "https://desk-cache.cachix.org" ];` and **dropping** `cache.nixos.org`. Every `hello` is now a local compile unless your team cache mirrors the world.

The boring rule

- `cache.nixos.org` plus **one** team cache. Not five random community caches you do not audit. Keep cache.nixos.org in the list.
- Public keys next to substituter URLs, in the daemon's `nix.conf`. `require-sigs = true`.
- CI pushes. Laptops pull. Do not require every laptop to upload.
- Never commit signing secrets. `nixConfig` in the flake is not the daemon.
- Confirm a hit with "copying path from …" before you call the cache done.

Try this

1. `nix show-config | grep -E 'substituters|trusted-public-keys'` and identify which keys match which URLs.
2. `nix build nixpkgs#hello --max-jobs 0` (no local build). It should substitute. If it errors, your substituter list is empty.
3. Create a `file://` cache of `hello` (Case 4) and copy it into a second store (`nix copy --from …`).
4. Intentionally typo the public key, rebuild a package you do not have locally, and read the error. Restore the key.

GitHub Actions with Nix

GitHub Actions with Nix

`actions/setup-go` plus `actions/setup-python` plus a matrix of cache keys is how CI drifts from laptops. The boring default is: **install Nix on the runner, `nix flake check` / `nix build`, and push results to the team binary cache**.

Mental model

The runner starts empty. You give it Nix, the same `flake.lock` as the laptop, and a cache token. The graph that `nix develop` used locally is the graph CI builds. There is no second `Dockerfile` for CI.

Step	Job
checkout	The flake and the lock
install Nix	Daemon + flakes
cache auth	Cachix (or magic-nix-cache for GHA's own cache)
`nix flake check`	Eval + checks + tests
`nix build`	The artifacts you ship

Use **pinned action SHAs or release tags**, not `@main`, once the workflow is real. Examples below use tags for readability.

Worked examples

Case 1: Check and build, push to Cachix

Save as `.github/workflows/ci.yml`:

```yaml
# .github/workflows/ci.yml
name: CI
on:
  pull_request:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
        with:
          extra_nix_config: |
            extra-substituters = https://desk-cache.cachix.org
            extra-trusted-public-keys = desk-cache.cachix.org-1:aaaaaaaaaaaaaaaaaaaaaaaaaaaa
      - uses: cachix/cachix-action@v15
        with:
          name: desk-cache
          authToken: ${{ secrets.CACHIX_AUTH_TOKEN }}
          skipPush: ${{ github.event_name == 'pull_request' }}
      - name: Check and build
        run: |
          nix flake check -L
          nix build --print-build-logs
          git diff --exit-code flake.lock
```

The flake lock already pins nixpkgs — `nix_path: nixpkgs=channel:…` does nothing for `nix flake check`. Skip it. `extra_nix_config` is how the **runner daemon** sees the team cache (same keys as `/etc/nix/nix.conf` on laptops).

`CACHIX_AUTH_TOKEN` is a repository secret. Pull requests from forks should **not** receive a write token (GitHub's default is correct: `pull_request` from forks cannot read org secrets). `skipPush` on `pull_request` is belt-and-suspenders when the workflow still sees the token on internal PRs.

Do not add `actions/setup-go` next to this job. The flake already has `go`.

If CI rewrote the lock, `git diff --exit-code flake.lock` fails. The lock in git is the pin; the runner does not get a vote.

```
# .github/workflows/ci.yml fragment
permissions:
  contents: read
concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true
```

`pull_request_target` is still forbidden with a write token (secrets chapter).

Case 2: `nix flake check` is the gate

Save as a flake fragment (merge into `outputs`):

```
# checks.nix - conceptually part of flake outputs
{
  checks.x86_64-linux.fmt = pkgs.runCommand "fmt" { } ''
    echo ok > $out
```

```
    '';
}
```

A failing check fails the job. You do not add a second "lint" job with a different Python.

```
nix flake check
```

Output on success:

```
evaluating flake...
checking flake...
```

Case 3: Magic Nix Cache when you do not have Cachix yet

```
# .github/workflows/ci-magic.yml
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: DeterminateSystems/nix-installer-action@v16
      - uses: DeterminateSystems/magic-nix-cache-action@v8
      - run: nix build
```

This uses GitHub's cache API. It is per-repository and evicts. It is a start. A team cache (Cachix/Attic) is what laptops share with CI.

Case 4: Pin the installer, pin nixpkgs

The flake lock already pins nixpkgs. The installer action should not float `@main`:

```
- uses: cachix/install-nix-action@v27
```

When you bump, bump in a dedicated commit. Same discipline as `flake.lock`.

Case 5: Free disk on the runner

GitHub's Ubuntu image is busy. If `nix build` dies with `No space left on device`:

```
- name: Free disk
  run: |
    sudo rm -rf /usr/share/dotnet /opt/ghc /usr/local/share/boost
    df -h
```

Then GC at the end of self-hosted runners only (`nix-collect-garbage -d`). Hosted runners die anyway.

The trap

The trap is **`nix-env -iA nixpkgs.go` in CI** after installing Nix, then `go test`. You just rebuilt the old world: an unpinned go, no flake, no cache sharing with the laptop.

The other trap is `cachix-action` with a write token on `pull_request_target`. A fork PR can then push malware NARs to the team cache. Write tokens: `push` to `main` and trusted workflows only.

A third: `nix_path: nixpkgs=channel:nixos-unstable` next to a 26.05 flake, then a step that uses `<nixpkgs>` — two pins, two worlds. A fourth: installing Nix twice (Cachix **and** Determinate) in one job.

The boring rule

- One install-Nix action, then `nix flake check` and `nix build`. `git diff --exit-code flake.lock`.
- Same `flake.lock` as developers. Team cache via `extra_nix_config`, not a channel `nix_path`.
- Push to the team cache from trusted branches (`skipPush` on `pull_request`).
- Pin action versions. One installer, not two.
- Do not `setup-go` / `setup-python` next to Nix "just in case."

Try this

1. Add Case 1 to a throwaway repo with a `flake.nix` that outputs `packages.x86_64-linux.default = pkgs.hello`. Confirm the Actions log shows a substitute or a build, then a second run that hits the cache.
2. Break `flake.nix` syntax, push, and confirm the job fails at eval, not at a later custom script.
3. List repository secrets and confirm `CACHIX_AUTH_TOKEN` is not printed in logs (`echo ${{ secrets.CACHIX_AUTH_TOKEN }}` is a firing offence).
4. Time `nix flake check` locally versus in CI. If CI is 10× slower, you are compiling; fix substituters (`extra_nix_config` keys).
5. Confirm the workflow YAML has no `pull_request_target` and no `nix_path` channel.

GitLab CI Integration

GitLab CI Integration

GitLab jobs are containers. The boring default is: **a Nix image (or a runner that already has Nix), flakes enabled, `nix flake check` / `nix build`, and a cache that is not `GIT_STRATEGY: none` plus hope**.

Mental model

Runner kind	How Nix appears
Shared SaaS runner	Job image `nixos/nix` (or `gitpod/workspace-nix`)
Self-hosted with Nix	`image:` optional; use the host store + a mounted `/nix`
Kubernetes executor	Persistent `/nix` volume or you download the world every job

`NIX_CONFIG` in `variables:` is the reliable way to turn on flakes inside `nixos/nix` images that ship a conservative `nix.conf`.

The same binary cache as GitHub Actions applies. Cachix works from GitLab. `CACHIX_AUTH_TOKEN` is a GitLab CI/CD variable, masked, protected if it can write.

Worked examples

Case 1: Official Nix image, flakes on

Save as `.gitlab-ci.yml`:

```yaml
# .gitlab-ci.yml
image: nixos/nix:2.35.2

variables:
  NIX_CONFIG: "experimental-features = nix-command flakes"

stages:
  - test
  - build
```

```yaml
test-job:
  stage: test
  script:
    - nix flake check

build-job:
  stage: build
  script:
    - nix build --print-build-logs
    - nix path-info -Shr ./result
```

Pin the image tag (`2.35.2`), not `latest`.

Case 2: Cachix from GitLab

```yaml
# .gitlab-ci.yml
image: nixos/nix:2.35.2

variables:
  NIX_CONFIG: "experimental-features = nix-command flakes"

before_script:
  - nix-env -iA nixpkgs.cachix
  - cachix use desk-cache
  - echo "$CACHIX_AUTH_TOKEN" | cachix authtoken --stdin

build:
  script:
    - nix flake check
    - nix build
    - cachix push desk-cache ./result
```

`nix-env -iA` for **cachix the client** on a throwaway image is acceptable. Do not use it to install the application under test.

A cleaner variant: `nix run github:NixOS/nixpkgs/nixos-26.05#cachix -- use desk-cache`.

Pin the image by digest when the runner can pull it:

```yaml
image: nixos/nix:2.35.2@sha256:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

`GIT_STRATEGY: clone` plus a shallow clone is fine. `GIT_STRATEGY: none` plus a mystery workspace is how you test last week's `flake.lock`.

Case 3: Cache `/nix` on a self-hosted runner

On a runner VM that keeps its disk:

```yaml
# .gitlab-ci.yml
default:
  tags: [nix-runner]
  image: nixos/nix:2.35.2
  cache:
    key: nix-store
    paths:
      - /nix/store
```

GitLab's `cache: paths` on `/nix/store` is **slow and incomplete** (no `db.sqlite`). Prefer a **persistent runner** with Nix installed on the host and:

```yaml
# .gitlab-ci.yml
default:
  tags: [nix-host]
  # no image: - job runs on the host
variables:
  NIX_CONFIG: "experimental-features = nix-command flakes"
build:
  script:
    - nix flake check
    - nix build
```

The host store **is** the cache. GC with `--delete-older-than 7d` in a nightly cron.

Case 4: `rules` so forks do not push

```yaml
# .gitlab-ci.yml
push-cache:
  script:
    - nix build
    - cachix push desk-cache ./result
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Write tokens stay on the default branch.

Case 5: Show what you shipped

```yaml
# .gitlab-ci.yml
build:
  script:
    - nix build
    - nix path-info -Shr ./result
    - nix-store -q --tree ./result | head -40
```

```
  artifacts:
    paths:
      - result
    expire_in: 1 day
```

The `result` symlink as an artifact is **not** a closure. It is a pointer that will not work on a machine without `/nix`. Use `nix copy --to file://$PWD/nar` and artifact `nar/` if humans must download.

The trap

The trap is **`image: ubuntu:24.04` plus `curl | sh` the Nix installer on every job**. You pay minutes and get a different Nix version than last week. Use a Nix image or a Nix-equipped runner.

The other trap is caching only `result` in GitLab artifacts and calling it a binary cache. Artifacts are not signed NARs.

The boring rule

- Pin `nixos/nix:<version>` (digest if you can) or use a self-hosted runner that already has Nix.
- `nix run nixpkgs#cachix`, not `nix-env` for the app under test.
- `NIX_CONFIG` enables flakes inside the job.
- Same team cache as GitHub Actions and laptops.
- Write tokens only on the default branch.
- `nix path-info -Shr` in the log so a fat closure is visible in review.

Try this

1. Run Case 1 against a flake that builds `hello`. Confirm the job log contains `Hello` or a store path for hello.
2. Pin the image digest (`nixos/nix:2.35.2@sha256:…`) and record why that is stricter than a moving tag.
3. Add `nix path-info -Shr ./result` and paste the closure size into the merge request template once.
4. On a self-hosted runner, run `nix-collect-garbage --delete-older-than 7d` after a week and note `df -h /nix`.

Build Caching Strategies

Build Caching Strategies

CI that recompiles llvm because someone edited `README.md` is not a pipeline, it is a heater. The boring default is: **substituters first, a persistent team cache second, GitHub's cache API only as a bump, and `src` filters so docs do not bust compiles**.

Mental model

Nix does not cache "step 3 of a Dockerfile." It caches **store paths**. If the `.drv` hash is unchanged, the path is substituted. If `src` includes the whole git tree, every commit is a new `.drv`.

Layers, cheapest first:

Layer	Lifetime	Who uses it
Local `/nix/store`	Until GC	This runner / laptop
Team cache (Cachix / Attic)	Months	Every laptop + every CI job
`magic-nix-cache` (GHA)	Days, per repo	GitHub-hosted runners only

Docker layer cache busts everything after a changed line. Nix rebuilds only the node whose inputs moved, plus dependents. A docs derivation and a Go derivation are siblings, not a stack.

```
README.md change
      hello-docs.drv   rebuilt
      desk-api.drv     same hash → substitute
```

Worked examples

Case 1: Team cache is the real cache

The binary-cache chapter already added substituters. CI must **push** what it built:

```yaml
# .github/workflows/ci.yml
name: CI
on:
  pull_request:
  push:
    branches: [main]
```

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - uses: cachix/cachix-action@v15
        with:
          name: desk-cache
          authToken: ${{ secrets.CACHIX_AUTH_TOKEN }}
      - run: nix build --print-build-logs
```

A laptop with the same `flake.lock` then substitutes. `magic-nix-cache` alone does **not** help the laptop.

Case 2: magic-nix-cache as a spare tyre

```yaml
# .github/workflows/ci-magic.yml
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: DeterminateSystems/nix-installer-action@v16
      - uses: DeterminateSystems/magic-nix-cache-action@v8
      - run: nix build
```

Fine for a personal repo. Evicts. Does not sign NARs for other machines. Graduate to Cachix when a second human waits on CI.

Case 3: `lib.cleanSource` / gitignore

Save as `package.nix`:

```nix
# package.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.stdenv.mkDerivation {
  pname = "desk-banner";
  version = "1.0.0";
  src = pkgs.lib.cleanSource ./.;
  buildPhase = "echo ok > $out";
  installPhase = "true";
```

```
}
```

`cleanSource` drops `.git`, `result`, and editor junk. Stricter:

```
src = pkgs.lib.cleanSourceWith {
  src = ./.;
  filter = path: type:
    baseNameOf path != "README.md"
    && baseNameOf path != "docs";
};
```

Or `pkgs.nix-gitignore.gitignoreSource [ ] ./.` with a `.gitignore` you actually maintain.

```
nix-build package.nix
echo "# note" >> README.md
nix-build package.nix
```

If the second build is "checking paths" / a cache hit, the filter worked. If it rebuilds, `README.md` is still in `src`.

Case 4: Vendor once, compile often

Save as `go.nix`:

```
# go.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = pkgs.lib.cleanSource ./.;
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

`vendorHash` is a FOD. Changing `main.go` does not re-fetch modules. Changing `go.mod` does. Probe the hash once; do not set `vendorHash = null` in CI "to make it work."

Case 5: See what would rebuild

```
nix build --dry-run
nix path-info -Shr ./result | tail -1
```

`--dry-run` lists derivations that are not in the store or substituters. If that list includes `gcc` after a comment-only change, `src` is too wide or an input leaked (`currentTime`, impure env).

```
nix build .#desk-api --rebuild --dry-run 2>&1 | head
```

`--rebuild` pretends the path is missing. Use it to see the **graph**, not to heat the room. `nix why-depends` after a surprise rebuild tells you which input moved (often `flake.lock` or an unfiltered `src`).

The trap

The trap is **`src = ./.`** on a monorepo root for a leaf package. A frontend CSS change rebuilds the Go API. Filter to `./cmd/desk-api` or `fileset` (`lib.fileset.gitTracked`).

The other trap is treating GHA cache as the team cache. A new contributor's first `nix build` still compiles the world.

The boring rule

- Push to a signed team cache from trusted CI. Laptops pull.
- Filter `src`. Docs and `.git` are not compile inputs.
- Language vendor hashes (Go, Cargo, npm) are FODs. Pin them.
- `--dry-run` after a trivial edit. If gcc is in the list, stop and fix inputs.
- magic-nix-cache is optional. Substituters are not.
- `--dry-run` / `why-depends` before you blame the cache. Filtered `src` before you add another cache.

Try this

1. `nix path-info -Shr ./result` before and after a README-only commit. The store path of the binary should be identical if `src` is filtered.
2. `--dry-run` twice in a row with no edits. Second list should be empty (or only substitutes).
3. Break `vendorHash` by one character, watch the FOD mismatch, restore it.
4. Time `nix build` on a clean store with the team cache configured versus `nix build --option substituters ''`. The delta is the cache earning its keep.

Testing in CI Pipelines

Testing in CI Pipelines

Tests that need the office Wi-Fi are not tests. The boring default is: **hermetic `checkPhase` / `go test` inside the derivation, `nix flake check` on every PR against nixpkgs 26.05, and NixOS VM tests when a unit must listen**.

Mental model

The sandbox has no network. `go test` that downloads modules fails. `vendorHash` (FOD) already fetched modules. `npm test` that hits staging fails. Mock it.

Kind	Where
Unit	`doCheck` / `checkPhase` on the package
Lint	`checks.<system>.lint`
Integration	`testers.runNixOSTest` (needs KVM)
Gate	`nix flake check -L` in CI

GitHub Actions: install Nix 2.35 (or the 26.05-pinned Nix), then `nix flake check`. No `setup-go`. Staging probes are **deploy** jobs, not `checks`.

Worked examples

Case 1: Hermetic Go tests

Save as `service.nix`:

```
# service.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.buildGoModule {
  pname = "desk-service";
  version = "1.0.0";
  src = pkgs.lib.cleanSource ./.;
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  doCheck = true;
}
```

`buildGoModule` already runs `go test` when `doCheck` is true. Probe `vendorHash`. `null` is not a pin.

```
nix-build service.nix
```

Case 2: Flake checks

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-service checks";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
      pkg = pkgs.callPackage ./service.nix { };
    in
    {
      packages.${system}.default = pkg;
      checks.${system}.unit = pkg;
      checks.${system}.vet = pkgs.runCommand "vet" {
        nativeBuildInputs = [ pkgs.go ];
        src = ./.;
      } ''
        cd $src
        GOPROXY=off go vet ./...
        touch $out
      '';
    };
}
```

`GOPROXY=off` proves you are not downloading. If vet needs modules, use the same `buildGoModule` env.

```
nix flake check --print-build-logs
```

Case 3: CI workflow

Save as `.github/workflows/check.yml`:

```yaml
# .github/workflows/check.yml
name: check
on: [pull_request, push]
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - uses: cachix/cachix-action@v15
        with:
          name: desk-cache
          authToken: ${{ secrets.CACHIX_AUTH_TOKEN }}
          skipPush: ${{ github.event_name == 'pull_request' }}
      - run: nix flake check -L
```

Push cache on `main` only. `CACHIX_AUTH_TOKEN` is a GitHub secret, not a flake input.

Case 4: VM test job (KVM)

```nix
# test-nginx.nix fragment in checks
{
  checks.x86_64-linux.nginx = pkgs.testers.runNixOSTest {
    name = "desk-nginx";
    nodes.machine = { services.nginx.enable = true; };
    testScript = ''
      machine.wait_for_unit("nginx.service")
      machine.succeed("curl -f http://127.0.0.1/")
    '';
  };
}
```

```
nix build .#checks.x86_64-linux.nginx -L
```

GitHub hosted runners often lack nested KVM. Use a self-hosted runner with `/dev/kvm` or skip VM tests on GHA and run them on the builder.

Case 5: Fail the gate

```
# break a test, then:
nix flake check -L
echo $?
```

Non-zero. The PR does not merge. Logs are in `nix log` / `-L`, not in a screenshot. Run check twice: the second should substitute.

The trap

The trap is `go test that hits https://staging.desk.internal`. Sandbox: connection refused. "Works on my laptop" because your laptop is not a sandbox. `doCheck` must be hermetic. Integration against staging is a **deploy** job with secrets, not `flake check`.

The boring rule

- `nix flake check -L` is the PR gate. 26.05 pin. Nix 2.35 on the runner.
- `vendorHash` / `cargoHash` so tests do not download.
- VM tests need KVM; do not pretend GHA ubuntu has it.
- No prod credentials in check.
- `-L` so failures are in the log.

Try this

1. Flip an assertion, `nix flake check`, restore.
2. Add `curl https://example.com` to `checkPhase` and confirm sandbox failure.
3. `nix flake check` twice; second should substitute.
4. Document whether VM tests run on GHA or on the builder.

Secrets Management in CI

Secrets Management in CI

The store is world-readable. CI logs are copied. Fork pull requests are hostile. The boring default is: **no secret bytes in Nix; `access-tokens` for private inputs; Age/sops keys only on trusted jobs; never `pull_request_target` plus a write token**.

Mental model

Kind of secret	Where it lives	How Nix sees it
GitHub PAT for private flakes	Runner `nix.conf` `access-tokens`	Fetch, not a derivation input
Age private key for sops	Runner env / `SOPS_AGE_KEY`	Decrypt at **eval/deploy** time
DB password for the app	Encrypted in git; decrypted on the **host**	Path `/run/secrets/…` only
Cachix write token	Actions secret	`cachix push`, not `flake.nix`

Builds should not need production passwords. If `nix flake check` requires Vault, the check is not hermetic.

```
trusted push to main
    → runner gets tokens
    → nix build   (no app secrets)
    → deploy      (decrypt on target, not in the NAR)
```

Worked examples

Case 1: Private flake inputs without putting the PAT in Nix

Save as `.github/workflows/deploy.yml`:

```yaml
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]

jobs:
```

```
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
        with:
          extra_nix_config: |
            access-tokens = github.com=${{ secrets.GH_PAT_FOR_PRIVATE_MODULES }}
      - run: nix flake check
      - run: nix build
```

`access-tokens` is daemon config. The token is not hashed into `desk-api`. Fork PRs must **not** receive this secret (GitHub's default for `pull_request` from forks).

Do **not** put the PAT in `flake.nix`:

```
# flake.nix - do not
{
  nixConfig.access-tokens = { github.com = "ghp_…"; };
}
```

That file is world-readable in git and in the eval. `nixConfig.extra-substituters` without matching `trusted-public-keys` is a different bug (unsigned cache, or compile-the-world).

Prefer GitHub **OIDC** (`id-token: write`) to a Cachix/cloud role over a long-lived PAT in Actions secrets, when the other side supports it. The PAT in Case 1 is for private flake inputs that still require one.

```
# .github/workflows/deploy.yml fragment
permissions:
  contents: read
  id-token: write
```

Default `GITHUB_TOKEN` is enough to checkout **this** repo. A PAT is only for **other** private GitHub flake inputs. Scope it `repo:read` on those repos, not `admin:org`.

Case 2: Eval sops without decrypting values into the store

```
# .github/workflows/sops-eval.yml
name: Sops eval
on:
  push:
    branches: [main]
jobs:
  eval:
    runs-on: ubuntu-latest
    steps:
```

```
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - name: Eval secret *paths*
        env:
          SOPS_AGE_KEY: ${{ secrets.CI_AGE_PRIVATE_KEY }}
        run: nix eval .#nixosConfigurations.desk-server.config.sops.secrets
```

The eval prints **paths** (`/run/secrets/db-password`), not password strings. If the log shows a password, the module leaked it into config (usually `hashedPassword = "plaintext"` or `environment.FOO = config.sops.secrets.x.path` is fine; interpolating `builtins.readFile` is not).

Case 3: The leak, on purpose, once

Save as `leak.nix`:

```
# leak.nix
{ pkgs ? import <nixpkgs> {} }:

pkgs.writeText "database.conf" ''
  password = SuperSecretPassword123
''
```

```
nix-build leak.nix
cat result
nix path-info result
```

Output:

```
password = SuperSecretPassword123
/nix/store/…-database.conf
```

Anyone with `nix copy` of this closure has the password. Delete the file, GC, **rotate** the password. Then never do this.

Case 4: Cachix write only on main

```
# .github/workflows/push-cache.yml
name: Push cache
on:
  push:
    branches: [main]
jobs:
  push:
    runs-on: ubuntu-latest
```

```yaml
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - uses: cachix/cachix-action@v15
        with:
          name: desk-cache
          authToken: ${{ secrets.CACHIX_AUTH_TOKEN }}
      - run: nix build
```

Do not use `pull_request_target` with this token. That event runs **your** workflow file from the base branch with **secrets**, but checks out (if you are not careful) or executes code from the fork. A fork PR can `cachix push` garbage — or steal the token from the log / a `curl | sh` step.

Event	Secrets	Runs whose YAML?
`pull_request` (fork)	No (GitHub default)	Merge of base + fork
`push` to `main`	Yes	Trusted
`pull_request_target`	**Yes**	Base workflow, attacker-controlled checkout if you `ref: ${{ github.event.pull_request.head.sha }}`

`pull_request_target` plus `actions/checkout` of the PR head plus a write token is a compromise of the cache and of any deploy key. Use `pull_request` for build checks; deploy only on `push` to `main`.

Case 5: `nix flake check` without credentials

```
unset SOPS_AGE_KEY CACHIX_AUTH_TOKEN GH_TOKEN
nix flake check
```

If this fails because sops cannot decrypt, split checks: eval that only looks at **paths** and types, versus a deploy job that has keys. Checks that need production secrets will fail on contributor laptops too.

The trap

The trap is `builtins.readFile ./prod.key` in a module. Eval copies the bytes into the store. `nix why-depends /run/current-system` will show a path whose `cat` is the key.

The other trap is printing secrets in `set -x` bash. `echo "$SOPS_AGE_KEY"` in a workflow is a rotation event.

A third is `pull_request_target` + **Cachix write / deploy key**. A fourth is `nixConfig.access-tokens` committed in the flake.

The boring rule

- Store is public. GitHub logs are public-ish. Treat both as hostile.
- `access-tokens` in **runner** `nix.conf` (Actions `extra_nix_config`), never in `flake.nix`.
- Encrypted files in git for app secrets. Decrypt on the **host**.
- Write tokens (Cachix, deploy keys) only on `push` to `main`. Never `pull_request_target`.
- `nix flake check` without production credentials.
- After any leak, rotate. GC does not un-copy a cache.

Try this

1. Case 3 on a throwaway password. `cat $(nix-build leak.nix --no-out-link)`. Then GC and confirm you still understand copies in the cache if you had pushed.
2. `nix-instantiate --eval -E 'builtins.readFile ./test.txt'` with a dummy file. Note that the string is in the eval result.
3. Confirm `nix flake check` on a clone with no `SOPS_AGE_KEY`.
4. Open the Actions secret list and tick "review logs for accidental echo" on the last deploy job.
5. Search the repo for `pull_request_target` and for `nixConfig.access-tokens`. Both should be empty.

Remote Builders

Remote Builders

Binary caches move **results**. Remote builders move **work**. The boring default is: **evaluate on the laptop, realise on a machine that has CPU, KVM, and the right `system`, then push the result to the team cache**.

A laptop should not compile Chromium. A Darwin laptop cannot natively produce `x86_64-linux` test VMs. Both problems are remote builders, not "more `-j`".

Mental model

	Binary cache	Remote builder
Needs the path to already exist?	Yes	No
What moves first	NAR (result)	`.drv` + input paths, then NAR back
Trust	Signing keys	SSH user on the builder
Typical role	Everyone pulls	One beefy box (or a queue) builds

```
laptop  --drv-->  builder  --push-->  cache  --substitute-->  fleet
```

`nix.buildMachines` (NixOS) or `--builders` (CLI) lists machines:

- `hostName`, `sshUser`, `sshKey`
- `system` (`x86_64-linux`, `aarch64-linux`, …)
- `protocol` (`ssh-ng` preferred)
- `maxJobs`, `speedFactor`
- `supportedFeatures` (`kvm`, `big-parallel`, `nixos-test`)

If a derivation requires `kvm` and the builder does not advertise it, Nix builds locally (or fails). Mysterious local compiles are often a missing feature flag.

`ssh-ng` is the Nix 2 protocol (one SSH session, multiplexed). Plain `ssh` (legacy) copies NARs over a new connection per path — slower, more fragile. Prefer `protocol = "ssh-ng"`. `mandatoryFeatures` is the strict sibling of `supportedFeatures`: the derivation **must** have those features or Nix will not pick the machine. Use it so a `big-parallel` Chromium job cannot land on a 2-core builder that merely listed `kvm`.

Worked examples

Case 1: NixOS client configuration

Save as `builders.nix`:

```
# builders.nix
{
  nix.distributedBuilds = true;
  nix.settings.builders-use-substitutes = true;

  nix.buildMachines = [
    {
      hostName = "builder.desk.internal";
      system = "x86_64-linux";
      protocol = "ssh-ng";
      sshUser = "nixbuilder";
      sshKey = "/root/.ssh/id_builder";
      maxJobs = 8;
      speedFactor = 4;
      supportedFeatures = [ "nixos-test" "big-parallel" "kvm" ];
    }
  ];
}
```

`builders-use-substitutes = true` lets the **builder** download from the team cache instead of asking the laptop to upload gcc.

The SSH key is a host key for root (the daemon builds as root's SSH). Protect it with sops. `nixbuilder` on the far side must be a trusted Nix user.

```
sudo nixos-rebuild switch
```

Case 2: One-shot without a module

```
nix build .#desk-api --max-jobs 0 \
  --builders 'ssh-ng://nixbuilder@builder.desk.internal x86_64-linux - 8 1 kvm,big-parallel,
```

`--max-jobs 0` means "do not build locally." If the builder is down, the command fails instead of melting the laptop.

Case 3: Builder host — NixOS

Save as `builder-host.nix`:

```
# builder-host.nix
{ pkgs, ... }:

{
  users.users.nixbuilder = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    openssh.authorizedKeys.keys = [
      "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskLaptopRootBuilderKey desk-laptop"
    ];
  };

  nix.settings.trusted-users = [ "root" "nixbuilder" ];
  nix.settings.max-jobs = 8;

  services.openssh.enable = true;
}
```

`trusted-users` is required for `nixbuilder` to start builds as the daemon. Do not put your whole team in `trusted-users` on a laptop; do it on the builder.

Test SSH **as the key the daemon will use**:

```
sudo ssh -i /root/.ssh/id_builder nixbuilder@builder.desk.internal nix-store --version
```

If this asks for a password, the daemon will hang.

Case 4: Darwin → Linux

An Apple silicon laptop producing Linux containers:

```
# darwin-builders.nix
{
  nix.distributedBuilds = true;
  nix.buildMachines = [
    {
      hostName = "linux-builder.desk.internal";
      system = "aarch64-linux";
      protocol = "ssh-ng";
      sshUser = "nixbuilder";
      sshKey = "/var/root/.ssh/id_builder";
      maxJobs = 8;
      supportedFeatures = [ "kvm" "big-parallel" "nixos-test" ];
    }
  ];
}
```

`linux-builder` can be a VM on the Mac (`nix.linux-builder.enable` on nix-darwin) or a real ARM server. Prefer a real server for `nixosTest` (KVM). The VM is fine for `GOARCH=arm64` CGO-off Go; it is a poor Hydra.

Root on the laptop does not read `~deskadmin/.ssh/config`. Put `Host builder` in `/root/.ssh/config` or use an IP in `hostName`.

Case 5: Features — why tests ran locally

```
nix build .#checks.x86_64-linux.desk-vm --max-jobs 0
```

If the check is a `nixosTest`, the builder **must** list `kvm` and `nixos-test`. Missing those, Nix tries the local machine; on Darwin that is a slow VM or a failure.

```
nix show-config | grep builders
```

Confirm the live config, not just the `.nix` file you meant to switch.

The trap

The trap is **pointing `buildMachines` at a machine that is also a substituter with a different nixpkgs**. You upload inputs, it rebuilds everything because its store is on 23.11. Same `flake.lock` everywhere; builder substitutes from the team cache (`builders-use-substitutes`).

The other trap is user SSH config (`~/.ssh/config` Host aliases) that root's daemon does not read. Use `hostName` IPs or put the alias in `/root/.ssh/config`.

The boring rule

- Cache for results, builders for first compiles and foreign `system`.
- `ssh-ng`, `builders-use-substitutes`, `--max-jobs 0` when you mean "remote only."
- Advertise `kvm` / `nixos-test` / `big-parallel` honestly.
- Daemon SSH keys, not your interactive `ssh-agent`, unless you designed for that.
- Builder nixpkgs + cache = laptop flake lock + team cache.

Try this

1. `sudo ssh` from Case 3 until `nix-store --version` prints without a password.
2. `nix build nixpkgs#hello --max-jobs 0 --builders '…'` and watch the builder's `nix-daemon` log (`journalctl -fu nix-daemon`).
3. Remove `kvm` from `supportedFeatures`, run a `nixosTest`, and note where it builds. Put `kvm` back.
4. After a remote build, `nix path-info -Shr ./result` on the laptop — the NAR should have been copied back. Then `cachix push` so the next laptop never builds it.

Containers and Kubernetes

Creating Container Images with Nix (dockerTools)

Creating Container Images with Nix (dockerTools)

`FROM debian:latest` plus `apt-get` is a different closure every Tuesday. The boring default is: **`pkgs.dockerTools.buildLayeredImage` (or `streamLayeredImage`) from nixpkgs 26.05, no daemon at build time, `cacert` if you speak TLS**.

Mental model

Nix builds an OCI tarball from store paths. Layers split by derivation so glibc stays cached when `desk-api` changes. No `/bin/bash` unless you listed `pkgs.bash`.

Function	Output
`buildLayeredImage` `streamLayeredImage`	`.tar.gz` you `docker load` / `podman load` a **script** that writes the tarball to stdout (`./result | podman load`) — smaller NAR in CI

`contents` is **deprecated**. Use `copyToRoot`. `fakeNss` is `/etc/passwd` + `/etc/group` + nsswitch (root + nobody). Without it, `User = "65534"` fails with "no such user". `extraCommands` runs with the image root as cwd (`mkdir -p tmp && chmod 1777 tmp`). `pkgs.iana-etc` if you see `getProtocolByName: no such protocol name: tcp`.

Podman and Docker both load the archive. Building does **not** need root or a daemon.

```
desk-api drv  +  cacert  →  layered OCI  →  docker load / podman load
```

Worked examples

Case 1: Layered image

Save as `image.nix`:

```
# image.nix
{ pkgs ? import <nixpkgs> {} }:

let
```

```
  deskApp = pkgs.writeShellApplication {
    name = "desk-api";
    text = ''
      echo "desk-api listening (fake)"
    '';
  };
in
pkgs.dockerTools.buildLayeredImage {
  name = "desk-api";
  tag = "1.0.0";
  copyToRoot = [
    deskApp
    pkgs.cacert
    pkgs.dockerTools.fakeNss
  ];
  extraCommands = ''
    mkdir -p tmp
    chmod 1777 tmp
  '';
  config = {
    Cmd = [ "${deskApp}/bin/desk-api" ];
    ExposedPorts."8080/tcp" = { };
    Env = [ "SSL_CERT_FILE=${pkgs.cacert}/etc/ssl/certs/ca-bundle.crt" ];
    User = "65534:65534";
  };
}
```

```
nix-build image.nix
podman load < result
podman run --rm desk-api:1.0.0
```

Output (shape):

```
desk-api listening (fake)
```

Prefer a version tag over `latest`. `User = "65534:65534"` is nobody from `fakeNss`. Numeric IDs skip a name lookup; still ship `fakeNss` so glibc `getpwuid` does not panic. The process cannot write `/` — that is the point. Writable dirs (`/tmp`, maybe a volume for `/var/lib/desk-api`) come from `extraCommands`. `extraCommands` runs at **image build** time as root in that fake root, not as `User` at runtime.

`streamLayeredImage` takes the **same** attr set as `buildLayeredImage`. The derivation is a script: `nix build .#imageStream && ./result | podman load`. CI prefers the stream so the NAR is the script, not a multi-hundred-megabyte tarball sitting in the cache twice.

Case 2: `vendorHash` for a real Go module

```
# image-go.nix
{ pkgs ? import <nixpkgs> {} }:

let
  deskApp = pkgs.buildGoModule {
    pname = "desk-api";
    version = "1.0.0";
    src = pkgs.lib.cleanSource ./.;
    vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  };
in
pkgs.dockerTools.buildLayeredImage {
  name = "desk-api";
  tag = "1.0.0";
  copyToRoot = [ deskApp pkgs.cacert pkgs.tzdata pkgs.dockerTools.fakeNss ];
  config = {
    Cmd = [ "${deskApp}/bin/desk-api" ];
    User = "65534:65534";
  };
}
```

Probe `vendorHash` once. `null` is not a pin.

Case 3: Inspect layers

```
tar -tf result | head
nix path-info -Shr result | tail -1
```

The image closure should be the app + libc + certs, not gcc. If gcc is in `path-info -r`, you copied `buildInputs` into `copyToRoot`. `contents = [ pkgs.stdenv ];` is the same bug under the old name.

Case 4: Flake package

One file returns the **attr set**, not an image. Both builders take that set:

```
# image-args.nix
{ pkgs, deskApp }:

{
  name = "desk-api";
  tag = "1.0.0";
  copyToRoot = [ deskApp pkgs.cacert pkgs.dockerTools.fakeNss ];
```

```
  extraCommands = ''
    mkdir -p tmp
    chmod 1777 tmp
  '';
  config = {
    Cmd = [ "${deskApp}/bin/desk-api" ];
    User = "65534:65534";
    Env = [ "SSL_CERT_FILE=${pkgs.cacert}/etc/ssl/certs/ca-bundle.crt" ];
  };
}
```

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
      deskApp = pkgs.writeShellApplication {
        name = "desk-api";
        text = ''echo "desk-api listening (fake)"'';
      };
      imageArgs = import ./image-args.nix { inherit pkgs deskApp; };
    in
    {
      packages.${system}.image = pkgs.dockerTools.buildLayeredImage imageArgs;
      packages.${system}.imageStream = pkgs.dockerTools.streamLayeredImage imageArgs;
    };
}
```

```
nix build .#image
skopeo copy docker-archive:./result docker://registry.desk.internal/desk-api:1.0.0
```

```
nix build .#imageStream
./result | podman load
# CI: stream to a temp archive, then skopeo (stdin support varies)
./result > /tmp/desk-api.tar
skopeo copy docker-archive:/tmp/desk-api.tar docker://registry.desk.internal/desk-api:1.0.0
```

CI does not start dockerd. `skopeo` talks to the registry. Pin the tag; do not push `latest` from `main` unless that is an explicit contract.

Case 5: Nobody is home in the image

```
podman run --rm --entrypoint /bin/sh desk-api:1.0.0
```

Output:

```
Error: no such file or directory
```

That is success for a production image. Debug with a **separate** image that includes `pkgs.busybox`. Do not ship busybox in the API tag.

The trap

The trap is **forgetting `cacert`**. HTTPS fails with `x509: certificate signed by unknown authority`. Include `pkgs.cacert` and `SSL_CERT_FILE` (or `dockerTools.caCertificates`).

The other trap is `copyToRoot = [ pkgs.stdenv ];` (or deprecated `contents`) so "debug is easy." You just shipped a compiler.

A third: **root in the image** because `fakeNss` was omitted and `User` was left unset. A fourth: no `/tmp` (mode `1777`) so the Go runtime cannot create scratch files.

The boring rule

- `copyToRoot`, not deprecated `contents`. `buildLayeredImage` or `streamLayeredImage` from the **same** args.
- Pin tags. `cacert` + `fakeNss` + `/tmp`. `User` is not root.
- `vendorHash` is a FOD. Probe once.
- Build in CI without dockerd. `skopeo copy` after.
- Debug images are a different output, not extra packages in prod.

Try this

1. Build Case 1, `tar -tf result | wc -l`.
2. Drop `cacert`, run a one-liner that `curl https://example.com` (add curl), read the TLS error, put cacert back.
3. `nix path-info -Shr result` and confirm gcc is absent.
4. Tag `1.0.0` and `1.0.1` after a script change; confirm the base layers' hashes stayed when only the top layer moved.
5. `podman inspect desk-api:1.0.0 --format '{{.Config.User}}'` — `65534:65534`. Drop `fakeNss`, run, read the getpwuid error, put it back.

Kubernetes Manifests as Nix Modules

Kubernetes Manifests as Nix Modules

Hand-written YAML that drifts from the image tag in CI is how Saturday pages start. The boring default is: **generate manifests from Nix (same pin as `.#image`), commit the generated YAML if the cluster cannot eval Nix, and skip Helm until a chart is forced on you**.

Mental model

Nix can produce JSON/YAML for `kubectl apply`. Libraries (`nixidy`, `kubenix`, or a 40-line `writeText`) all do the same job: **one evaluation, one tag, one registry**.

The cluster still runs kubelet. Nix does not replace Kubernetes. It replaces the YAML copypaste.

```
flake  →  image tag + digest
       →  Deployment YAML with that digest
       →  kubectl apply --server-side
```

If the platform already speaks Helm only, next chapter. If you own the app, raw manifests are enough. Secrets still do not belong in ConfigMaps (store-is-public).

Worked examples

Case 1: A Deployment as JSON

Save as `k8s.nix`:

```nix
# k8s.nix
{ pkgs ? import <nixpkgs> { } }:

let
  image = "registry.desk.internal/desk-api@sha256:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  deploy = {
    apiVersion = "apps/v1";
    kind = "Deployment";
    metadata = { name = "desk-api"; namespace = "desk"; };
    spec = {
      replicas = 2;
      selector.matchLabels.app = "desk-api";
      template = {
```

```
        metadata.labels.app = "desk-api";
        spec.containers = [
          {
            name = "desk-api";
            inherit image;
            ports = [ { containerPort = 8080; } ];
            resources.requests = { cpu = "50m"; memory = "64Mi"; };
            resources.limits = { memory = "128Mi"; };
            securityContext = {
              runAsNonRoot = true;
              runAsUser = 65534;
              allowPrivilegeEscalation = false;
              readOnlyRootFilesystem = true;
            };
            livenessProbe.httpGet = { path = "/healthz"; port = 8080; };
            readinessProbe.httpGet = { path = "/healthz"; port = 8080; };
          }
        ];
      };
    };
  };
  svc = {
    apiVersion = "v1";
    kind = "Service";
    metadata = { name = "desk-api"; namespace = "desk"; };
    spec = {
      selector.app = "desk-api";
      ports = [ { port = 8080; targetPort = 8080; } ];
    };
  };
in
pkgs.writers.writeJSON "desk-api.json" [ deploy svc ]
```

```
nix-build k8s.nix
python3 -m json.tool result | head
```

`writers.writeJSON` is the 26.05 helper (pretty JSON, one derivation). Pin **digest**, not `:latest`. `image: registry/desk-api:1.0.0@sha256:…` (tag + digest) is accepted by kubelet and still human-readable. Every node may otherwise run a different blob.

`runAsUser = 65534` must match the image (`fakeNss` nobody). A probe on `/healthz` that the binary does not serve is a permanent `NotReady`.

Case 2: Same flake as the image

Save as `flake.nix`:

```
# flake.nix
{
  description = "desk-api image + manifests";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      packages.x86_64-linux.manifests = pkgs.callPackage ./k8s.nix { };
    };
}
```

```
nix build .#manifests
```

Bump the digest in **one** place and rebuild image + manifests together.

Case 3: Apply

```
kubectl apply --server-side --dry-run=server -f result
kubectl apply --server-side -f result
kubectl -n desk rollout status deploy/desk-api
```

`--server-side` plays nicer with fields you do not own (HPA replicas, kubectl last-applied). Dry-run first on a shared cluster. `--force-conflicts` is for when you **mean** to take a field back from another manager — not a default.

Case 4: ConfigMap from Nix, not from kubectl create

```
# configmap.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.writeText "desk-cm.json" (builtins.toJSON {
  apiVersion = "v1";
  kind = "ConfigMap";
  metadata = { name = "desk-api"; namespace = "desk"; };
  data.WINDOW = "A";
})
```

`WINDOW` is not a password. Use External Secrets / sealed-secrets / CSI for those — or do not put the password in git, same rule as the Nix store.

Case 5: When the cluster cannot run Nix

```
nix build .#manifests
cp --no-preserve=mode "$(readlink -f result)" generated/desk-api.json
git add generated/desk-api.json
git diff --exit-code generated/
```

CI diffs `generated/` so humans cannot `kubectl edit` without updating the flake. GitOps (Argo/Flux) applies the committed JSON.

The trap

The trap is **templating the image as `latest`**. Every node may run a different digest. Pin tag **and** digest (`image: repo:tag@sha256:…`) when the registry supports it.

The other trap is a Secret manifest with `data: { password: hunter2 }` in the flake. That is Case 1 of the secrets chapter, in YAML.

A third: `runAsNonRoot` on an image that still runs as uid 0 and has no `fakeNss` — the pod is `CreateContainerConfigError`. Build the image and the manifest from the **same** flake.

The boring rule

- Manifests and image from one flake pin (nixpkgs 26.05). `writers.writeJSON`.
- Digest (and a tag), not `latest`. Probes and `runAsNonRoot` match the image.
- Apply `--server-side`. Dry-run first.
- No passwords in ConfigMaps or Secret YAML in git.
- Generate-and-commit if GitOps tools cannot eval Nix.
- Helm next chapter, only when the vendor ships a chart.

Try this

1. Case 1, `jq '.[0].spec.replicas' result`.
2. Change replicas to 3, rebuild, `diff` the JSON.
3. Put `image: …:latest` in a lab, list why you will not ship it.
4. Add the Service (already in Case 1) and `kubectl apply --dry-run=server`.

Helm Charts Generated from Nix

Helm Charts Generated from Nix

Helm is the cluster's package manager whether you like it or not. The boring default is: **pin the chart *and* the values in Nix (vendored `Chart.yaml` tarball + hash), render with `helm template`, and never `helm install` from floating `latest`.**

Mental model

A chart is a tarball plus values. Unpinned `helm repo add && helm upgrade` is `npm install` on the control plane.

Pin	How
Chart version	`fetchurl` of the `.tgz` + hash (FOD)
Values	Nix attrset → YAML
Image inside values	Same digest as `.#image`

If the app is yours, prefer raw manifests. Helm when the vendor only ships a chart (ingress-nginx, cert-manager). `helm repo update` in CI is how Monday's job is not Friday's chart.

Worked examples

Case 1: Vendor a chart tarball

```
helm pull ingress-nginx --version 4.11.0 --repo https://kubernetes.github.io/ingress-nginx
nix hash path ingress-nginx-4.11.0.tgz
```

Save as `chart.nix`:

```
# chart.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.fetchurl {
  url = "https://github.com/kubernetes/ingress-nginx/releases/download/helm-chart-4.11.0/ing
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build chart.nix
```

Probe the hash (`got:`). That tarball is now as pinned as `hello`. Commit the hash, not a live repo index.

Case 2: Render values from Nix

Save as `values.nix`:

```
# values.nix
{
  controller = {
    replicaCount = 2;
    image.tag = "v1.11.2";
    service.type = "ClusterIP";
  };
}
```

Helm wants YAML. Do not hope JSON-as-values is always accepted — write YAML from Nix:

```
# values-file.nix
{ pkgs, ... }:

pkgs.writers.writeYAML "values.yaml" (import ./values.nix)
```

```
nix-build -E 'with import <nixpkgs> {}; callPackage ./values-file.nix {}'
helm template desk-ing ./ingress-nginx-4.11.0.tgz \
  --namespace desk \
  --include-crds \
  --kube-version 1.31.0 \
  -f result \
  > generated/ingress.yaml
```

`--include-crds` is how cert-manager actually installs. Without it you apply Deployments against CRDs the cluster does not have. `--kube-version` pins the Capabilities the chart sees so CI and the cluster do not render different `apiVersions`. `--namespace` must match GitOps `targetNamespace`.

Commit `generated/ingress.yaml` if Argo cannot run Helm+Nix. The image tag must match the digest your flake built, not `latest`.

Case 3: Wrapper script in a flake

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk helm render";
```

```
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
      chart = pkgs.fetchurl {
        url = "https://github.com/kubernetes/ingress-nginx/releases/download/helm-chart-4.11
        hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
      };
    in
    {
      packages.x86_64-linux.render-ing = pkgs.writeShellApplication {
        name = "render-ing";
        runtimeInputs = [ pkgs.kubernetes-helm ];
        text = ''
          helm template desk-ing ${chart} \
            --namespace desk \
            --include-crds \
            --kube-version 1.31.0 \
            -f ${./values.yaml}
        '';
      };
    };
}
```

`values.yaml` can be `builtins.toJSON` written by a derivation so it is not hand-edited. The chart path is a store path — no `helm repo add`.

Case 4: Apply the bits you reviewed

```
nix run .#render-ing | kubectl apply --server-side -f -
```

or `helm upgrade --install desk-ing ./ingress-nginx-4.11.0.tgz -f values.yaml` **from the vendored tgz**, not from a live repo index. GitOps (Argo/Flux) applies the rendered YAML; it does not call `helm repo update`.

Case 5: Diff in CI

```
nix run .#render-ing > /tmp/new.yaml
git diff --no-index generated/ingress.yaml /tmp/new.yaml
```

If the diff is unexpected (chart author pushed a hook), you see it before the cluster does. Non-zero diff on an unreviewed chart bump is a failed PR, not an automatic apply.

Pin the **image** inside values to a digest, same rule as raw manifests:

```nix
# values.nix fragment
{
  controller.image = {
    registry = "registry.k8s.io";
    image = "ingress-nginx/controller";
    tag = "v1.11.2";
    digest = "sha256:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
  };
}
```

The trap

The trap is **`helm repo update` in CI** with no version. Monday's job is not Friday's chart. Version + hash, like any FOD. The other trap is pinning the chart and leaving `image.tag: latest` in values.

A third: `helm template` **without** `--include-crds` for a CRD chart. A fourth: omitting `--kube-version` so a laptop on Helm 3.16 and CI on 3.17 emit different YAML for the same chart.

The boring rule

- Helm for vendor charts. Nix manifests for your apps.
- Chart tarball in the store (`fetchurl` + hash) or git-vendor.
- Values from Nix (`writers.writeYAML`). Images from the same flake (digest).
- `helm template --include-crds --kube-version … --namespace …`. Render in CI; apply the bits you reviewed.
- No `latest` chart, no `latest` image. No `helm repo update`.

Try this

1. `helm pull` a tiny chart, `nix hash path` the tgz, put it in `fetchurl`.
2. Change `replicaCount`, re-render, `diff`.
3. Break the hash by one character and read the FOD error.
4. Search CI for `helm repo update` and delete it.
5. Render twice, once with `--include-crds` and once without; `diff` — know what you would have skipped.

GitOps with Argo CD and Flux

GitOps with Argo CD and Flux

SSH-from-laptop `kubectl apply` does not scale and does not audit. The boring default is: **the cluster pulls generated YAML from git (Flux or Argo), and Nix only runs in CI to refresh that YAML**.

Mental model

Tool	Pull model
Flux	controllers reconcile `GitRepository` + `Kustomization`
Argo CD	Application CR points at a git path

Both want **plain YAML** in the repo (or a Helm release they render). They do not eval your flake on the control plane. CI:

```
nix build .#manifests  →  commit generated/  →  Flux applies
```

Pick **one**. Running Argo and Flux on the same cluster is a fight.

Worked examples

Case 1: Generated tree layout

```
generated/
  desk/
    namespace.yaml
    deploy.yaml
    svc.yaml
```

```
nix build .#manifests
rsync -a --delete result/ generated/desk/
git add generated && git diff --cached
```

Human reviews the YAML, not the Nix, at merge time — or reviews both.

Case 2: Flux GitRepository (illustrative)

Save as `generated/flux/gitrepo.yaml`:

```yaml
# gitrepo.yaml
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: desk
  namespace: flux-system
spec:
  interval: 1m
  url: https://git.desk.internal/platform/desk.git
  ref:
    branch: main
```

The URL is your repo. Auth is a Secret Flux already knows, not a token in Nix.

Case 3: Kustomization pointing at generated YAML

```yaml
# kustomization-flux.yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: desk-api
  namespace: flux-system
spec:
  interval: 5m
  sourceRef:
    kind: GitRepository
    name: desk
  path: ./generated/desk
  prune: true
  wait: true
  timeout: 5m
  targetNamespace: desk
```

`prune: true` deletes objects you removed from git. That is the point. Test in **staging** first: delete `deploy.yaml` in a branch that Flux watches on the staging cluster, `flux reconcile kustomization desk-api --with-source`, confirm the Deployment is gone, then merge the same prune policy to prod.

`wait: true` blocks the Kustomization until health checks pass. Without it, Flux marks the apply done while pods are still `CrashLoop`. `targetNamespace` keeps generated namespaced objects out of `flux-system`.

```
flux diff kustomization desk-api --path generated/desk
```

That diff is what on-call reads. If it wants to delete a CRD or a Namespace you still need, `prune` is too wide — split those objects into a Kustomization with `prune: false`.

Case 4: CI gate

```
# .github/workflows/manifests.yml
name: manifests
on: [pull_request]
jobs:
  gen:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - run: nix build .#manifests
      - run: diff -ru generated/desk result
```

If someone edited YAML by hand, the PR fails. Edit Nix, regenerate, commit both. `diff` against `result/` (the store path), not against a dirty working tree. Empty diff on a no-op commit is the contract.

Optional: `kubeconform` / `kubectl --dry-run=server apply -f` against staging as a second job. Nix generating YAML does not prove the API server will accept it.

Case 5: Argo Application (if the org standardised on Argo)

```
# application.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: desk-api
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://git.desk.internal/platform/desk.git
    targetRevision: main
    path: generated/desk
  destination:
    server: https://kubernetes.default.svc
    namespace: desk
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

```
    syncOptions:
      - CreateNamespace=true
```

Same generated path. Do not point Argo at `helm/` with unpinned deps. `selfHeal: true` reverts `kubectl edit` on the cluster — that is the GitOps contract, not a bug. Pin `targetRevision` to a **tag** for production if you need a freeze; `main` is a moving apply.

Turn prune on in the staging Application first. Argo's prune of a shared `Namespace` or a CRD is the same footgun as Flux.

Secrets in that tree stay encrypted. If you use KSOPS / Flux `sops-secrets`, the **cipher** is in git; the controller decrypts in-cluster. Plain `Secret` YAML in `generated/` is the store-is-public bug again.

```
# do not
apiVersion: v1
kind: Secret
stringData:
  password: hunter2
```

The trap

The trap is **the laptop as GitOps**: `flux reconcile` after `kubectl apply` after `nix build` that never got pushed. Git is the desired state. If it is not in `main`, the cluster should not have it.

The other trap is **`prune: true` first applied in production**. A deleted YAML is a deleted Deployment. Staging cluster, then prod. A third: committing plaintext `Secret` YAML (Case 5). A fourth: skipping the CI `diff` so generated/ and Nix drift.

The boring rule

- One GitOps controller. Generated YAML in git. Nix in CI.
- `diff -ru generated/ result` on every PR. Empty on no-ops.
- `prune` + `wait` in **staging** before production. Split CRDs/namespaces if prune would eat them.
- Secrets still not in that YAML (sops + KSOPS / External Secrets).
- Pin git `targetRevision` to a tag for prod if you need freeze.

Try this

1. `diff -ru` two `nix build .#manifests` trees with no edits — empty.
2. Change a replica count in Nix, regenerate, read the YAML diff as if you were on-call.
3. Write a one-page "we use Flux xor Argo" note in the desk repo.
4. Confirm `kubectl` on your laptop is **read** against prod, not write (RBAC).

5. On a lab cluster, enable prune, delete one generated manifest, reconcile, confirm the object is gone. Then restore it from git.

Service Mesh and Observability

Service Mesh and Observability

A mesh is another control plane. The boring default is: **Prometheus + logs + a NetworkPolicy before Istio; add a mesh only when mTLS between 20+ services is a written requirement**.

Mental model

Need	Boring tool
CPU/RAM/HTTP metrics	`prometheus` + `node_exporter` / app `/metrics`
Logs	journald on NixOS; Loki if you already have it
mTLS inside the cluster	Mesh (Istio/Linkerd) **or** app TLS
Traffic split	Ingress / Service, then mesh

NixOS can run Prometheus on the **ops** host without Kubernetes. In-cluster, same rule: pin charts, generate YAML.

This chapter does not install Istio. It tells you what to measure first so you do not. A mesh will not invent RED metrics for an app that has no `/metrics`.

Worked examples

Case 1: App metrics without a mesh

Save as `metrics.nix` (NixOS `ops-01`):

```
# metrics.nix
{
  services.prometheus = {
    enable = true;
    listenAddress = "10.100.0.30";
    port = 9090;
    scrapeConfigs = [
      {
        job_name = "desk-api";
        static_configs = [ { targets = [ "10.100.0.11:8080" ]; } ];
      }
      {
```

```
        job_name = "node";
        static_configs = [
          { targets = [ "10.100.0.11:9100" "10.100.0.12:9100" ]; }
        ];
      }
    ];
  };
  services.prometheus.exporters.node.enable = true;
  networking.firewall.interfaces.wg0.allowedTCPPorts = [ 9090 9100 ];
}
```

The Go service exposes `/metrics` on `8080`. Scrape only on `wg0`. Public 9090 is how you donate metrics to the internet.

Case 2: Probe the endpoint

```
curl -sS http://10.100.0.11:8080/metrics | head
```

Output (shape):

```
# HELP go_goroutines Number of goroutines
# TYPE go_goroutines gauge
go_goroutines 7
```

No output → fix the app, do not add Envoy. Then:

```
curl -sS http://10.100.0.30:9090/api/v1/targets | jq '.data.activeTargets[].health'
```

`up` is the boring dashboard. Grafana is optional.

Save a recording rule only when you have a query you already run by hand:

```
# prometheus-rules.nix fragment
{
  services.prometheus.rules = [
    ''
      groups:
        - name: desk-api
          rules:
            - alert: DeskApiDown
              expr: up{job="desk-api"} == 0
              for: 2m
    ''
  ];
}
```

An alert with no runbook in `docs/threats.md` is noise. Do not add twenty.

Case 3: NetworkPolicy before sidecar

Save as `netpol.yaml`:

```
# netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: desk-api
  namespace: desk
spec:
  podSelector:
    matchLabels:
      app: desk-api
  policyTypes: [Ingress]
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: desk-ing
      ports:
        - port: 8080
```

Generate this from Nix like any manifest. Most "we need a mesh" tickets are "we need NetworkPolicy." Apply, then `curl` from a pod that is **not** `desk-ing` and confirm it fails.

Case 4: If you must mesh

Pin the Linkerd or Istio **chart version + hash** (Helm chapter). Inject sidecars in **one** namespace first (`desk`). Measure CPU on the sidecar versus the app. Roll back with GitOps prune.

Do not enable the mesh globally on day one. Do not run Istio **and** Linkerd. Two meshes is two CNI races.

Case 5: Logs on NixOS already exist

```
journalctl -u desk-api -n 50 --no-pager
journalctl -u prometheus -n 20 --no-pager
```

Ship them with `services.promtail` or `systemd-journal-upload` if you have a Loki. A mesh access log is extra, not a substitute for application logs. `desk-api` returning 500 with a perfect Envoy dashboard is still a 500.

The trap

The trap is **installing Istio to "get observability."** You get a second failure domain, CNI races, and a dashboard. Start with `/metrics` and NetworkPolicy. The other trap is Prometheus on `0.0.0.0:9090` without a firewall — scrape configs are not a secret, but they describe your fleet.

The boring rule

- `/metrics` on the app. Prometheus scrape config in Nix. Listen on `wg0`.
- NetworkPolicy before sidecars.
- Mesh is optional and pinned. One namespace first. One mesh.
- journalctl is a valid log system.
- A mesh does not fix bad schema, slow SQL, or missing auth.

Try this

1. Hit `/metrics` on a desk service or add a one-line Prometheus handler in a lab binary.
2. Write a NetworkPolicy that allows only the ingress pods; prove a sidecar-less deny.
3. List three problems a mesh does **not** fix (bad schema, slow SQL, missing auth).
4. `nixos-option services.prometheus.scrapeConfigs` on 26.05 and add one job.

Infrastructure as Code

Managing Terraform Providers with Nix

Managing Terraform Providers with Nix

`terraform init` downloading whatever the registry serves this morning is a supply chain. The boring default is: **`pkgs.opentofu` or `pkgs.terraform.withPlugins` from nixpkgs 26.05 so providers are store paths**.

OpenTofu is the next chapter; the wrapper pattern is the same. Prefer OpenTofu on a new desk. This chapter names Terraform because that is the API people google.

Mental model

`withPlugins` builds a Terraform/OpenTofu **wrapper** whose plugin directory is in `/nix/store`. `init` does not need the network for providers. State still lives where you put it (S3, local) — Nix does not replace the backend.

```
nix develop  →  tofu/terraform in PATH
                 plugins = /nix/store/…-terraform-providers
terraform init -plugin-dir=…  →  offline providers
```

`which terraform` must be a store path. Distro `terraform` on PATH is how CI still hits the registry.

Worked examples

Case 1: Shell with pinned providers

Save as `terraform-env.nix`:

```
# terraform-env.nix
{ pkgs ? import <nixpkgs> { } }:

let
  tf = pkgs.terraform.withPlugins (p: [
    p.aws
    p.random
    p.local
  ]);
in
pkgs.mkShell {
```

```
  packages = [ tf ];
}
```

```
nix-shell terraform-env.nix --run "terraform version"
which terraform
```

Output (shape):

```
Terraform v1.x.x
on linux_amd64
+ provider registry.terraform.io/hashicorp/aws v…
+ provider registry.terraform.io/hashicorp/random v…
```

Versions are whatever 26.05 pinned. To freeze harder, overlay a specific provider derivation.

Case 2: Flake devShell

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk Terraform (prefer OpenTofu for new work)";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
      tf = pkgs.terraform.withPlugins (p: [ p.random p.local ]);
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell { packages = [ tf ]; };
    };
}
```

```
nix develop --command terraform version
```

Case 3: Offline init

```
nix develop --command terraform init -get=false
# PR job: no backend credentials required
nix develop --command terraform init -input=false -backend=false
nix develop --command terraform validate
```

If providers are already in the wrapper, init should not download. If it tries the registry, the wrapper is not on `PATH` (you ran distro terraform). Commit `.terraform.lock.hcl`. `-plugin-dir` is already baked into the wrapper; do not also set `TF_PLUGIN_CACHE_DIR` in `$HOME`.

```
readlink -f "$(which terraform)"
```

Must start with `/nix/store`.

Case 4: Tiny plan

Save as `main.tf`:

```
# main.tf
resource "random_id" "desk" {
  byte_length = 4
}

output "hex" {
  value = random_id.desk.hex
}
```

```
nix develop --command terraform init
nix develop --command terraform plan
nix develop --command terraform apply -auto-approve
```

No AWS credentials required. Proves plugins execute. Add `*.tfstate*` to `.gitignore`.

Case 5: CI without `setup-terraform`

Save as `.github/workflows/tf.yml`:

```
# .github/workflows/tf.yml
name: tf
on: [pull_request]
jobs:
  plan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: cachix/install-nix-action@v27
      - run: nix develop --command terraform fmt -check
      - run: nix develop --command terraform init -input=false -backend=false
      - run: nix develop --command terraform validate
      - run: nix develop --command terraform plan -input=false
        env:
          TF_IN_AUTOMATION: "1"
```

Same flake as laptops. No `hashicorp/setup-terraform`. Apply is a protected job on `main`, not this PR workflow.

The trap

The trap is **`terraform init` in CI with a clean `$HOME` and no `withPlugins`**. Registry outage or a yanked version becomes your outage.

The other trap is wrapping Terraform **and** OpenTofu on the same PATH. State gets written by the wrong binary. One CLI.

The boring rule

- Providers from nixpkgs (26.05 pin), not the registry at apply time. `withPlugins` + lockfile.
- `which terraform` is a store path.
- PR: `init -backend=false` + `validate`. Plan/apply need the backend.
- State backend is still your problem (lock table, encryption). Credentials are sops / env, not Nix strings.
- Prefer OpenTofu for new work (next chapter). One CLI on PATH.
- Overlay only when you must pin a provider the channel does not have.

Try this

1. Case 1, `which terraform`, confirm `/nix/store`.
2. `terraform init` with network blocked (`unshare -n` if you can) to confirm no download.
3. Add `p.null` to the plugin list, enter a new shell, `terraform version`.
4. `git grep setup-terraform` and delete it from CI.

OpenTofu Integration

OpenTofu Integration

Terraform's license change is why OpenTofu exists. The boring default for a **new** desk is: **`pkgs.opentofu` with `withPlugins` on nixpkgs 26.05, `.tf` files, same backend discipline**.

Mental model

OpenTofu is a Terraform-compatible CLI. Providers from nixpkgs attach the same way: `withPlugins` builds a wrapper whose plugin directory is in `/nix/store`. `tofu init` does not need the network for those providers. State still lives where you put it (S3, local) — Nix pins the **binary**, not the AWS account.

```
nix develop  →  tofu on PATH
                 plugins = /nix/store/…-opentofu-plugins
tofu init    →  offline providers
tofu plan    →  state from the backend
```

Commands: `tofu init|plan|apply|fmt`. Do not install both Terraform and OpenTofu on `PATH` unwrapped. State files from Terraform 1.x often import; test in a **clone** of state first.

Worked examples

Case 1: devShell with pinned plugins

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk OpenTofu";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
      tofu = pkgs.opentofu.withPlugins (p: [
        p.aws
        p.random
        p.local
```

```
      ]);
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
        packages = [ tofu ];
      };
    };
}
```

```
nix develop --command tofu version
```

Output (shape):

```
OpenTofu v1.x.x
on linux_amd64
```

The path is under `/nix/store`. `which tofu` must not be `~/.local/bin/tofu`.

`withPlugins` is the provider **binary**. You still commit `.terraform.lock.hcl` (hashes of those plugins). `tofu init -input=false` in this shell should **not** hit the registry; if it does, PATH is not the wrapper. `required_providers` in `.tf` documents versions; the store pin is what actually runs. Do not also set `TF_PLUGIN_CACHE_DIR` to a home directory that then races the wrapper.

Case 2: Same `main.tf` as Terraform

Save as `main.tf`:

```
# main.tf
resource "random_pet" "desk" {
  length = 2
}

output "name" {
  value = random_pet.desk.id
}
```

```
nix develop --command tofu init
nix develop --command tofu plan
nix develop --command tofu apply -auto-approve
```

No AWS required. Local state appears as `terraform.tfstate` — add `*.tfstate*` to `.gitignore`.

Case 3: Backend still not in Nix

Save as `backend.tf`:

```
# backend.tf
terraform {
  backend "s3" {
    bucket = "desk-tf-state"
    key    = "desk/opentofu.tfstate"
    region = "eu-central-1"
  }
}
```

Credentials for the backend come from the environment or sops (`AWS_PROFILE`, `AWS_ACCESS_KEY_ID`), not from a string in the flake. Nix does not substitute your account id.

```
nix develop --command tofu init -input=false
```

Case 4: CI splits plan and apply

```
nix develop --command tofu fmt -check
nix develop --command tofu init -input=false -backend=false   # fmt/validate, no state
nix develop --command tofu init -input=false                  # plan needs the backend
nix develop --command tofu validate
nix develop --command tofu plan -input=false -out=plan.bin
```

`-backend=false` is the PR job that must not need AWS credentials. Apply is a separate, protected job on `main` (environment protection, two reviewers) **with** the backend. Commit `plan.bin` nowhere. Artifact the plan between jobs if you must, with a short TTL. `TF_IN_AUTOMATION=1` turns prompts into errors — set it in CI.

Case 5: Do not mix CLIs

```
# in a desk-only shellHook, if humans still type terraform:
# alias terraform=tofu
# better: teach tofu
```

A shell that has both unwrapped is how state gets written by the wrong binary. `tofu state pull` after a surprise `terraform apply` is a long afternoon.

```
nix develop --command bash -c 'which tofu; command -v terraform || echo no-terraform'
```

`which tofu` is `/nix/store/…`. `terraform` should be missing, or a wrapper that *is* tofu. Two real binaries is the incident.

State migration: copy the backend key (`desk/opentofu.tfstate.migratetest`), `tofu init`, `tofu plan` against the copy, then cut. Not Friday in production.

The trap

The trap is **migrating state on Friday in production** without `tofu plan` against a copied state. The other trap is `terraform init` / `tofu init` downloading providers from the registry after you already pinned them in `withPlugins` — that means PATH is not the wrapper (`which tofu` is `~/.local` or a distro package).

A third: both `pkgs.terraform` and `pkgs.opentofu` on the same `PATH`. A fourth: committing `plan.bin` or `*.tfstate`.

The boring rule

- New work: OpenTofu from 26.05. `withPlugins`. Commit `.terraform.lock.hcl`.
- One CLI on PATH. Store path, not `~/.local`. No Terraform next to it.
- State credentials are secrets (sops / env), not Nix strings.
- PR: `fmt` + `init -backend=false` + `validate`. Apply on main with protection.
- Test state migration on a copy of the backend key.

Try this

1. `nix develop --command tofu version` and confirm a store path (`which tofu`).
2. `tofu plan` on `random_pet` without AWS.
3. `tofu fmt -check` on a badly indented file, fix, re-run.
4. Write down where state lives for the desk (bucket + key) in the repo README — not the credentials.
5. `nix develop --command tofu init -input=false` with `NIX_SSL_CERT_FILE` set and network blocked (`unshare -n` in a lab) — it must still find providers.

Ansible Provisioning with Nix

Ansible Provisioning with Nix

Ansible mutates hosts. NixOS **replaces** that for machines you own. The boring default is: **do not Ansible a NixOS box; if you must drive a fleet of Ubuntu leftovers, pin `pkgs.ansible` in a 26.05 devShell and keep inventory in git**.

Mental model

Host OS	Tool
NixOS	`nixos-rebuild` / Colmena / nixos-anywhere
Legacy Ubuntu/Debian you cannot reinstall this quarter	Ansible from a Nix shell

Using both on the same NixOS host (`ansible ad-hoc apt`) fights the next `switch`. The playbook that `copy`s a `configuration.nix` and then `shell: nixos-rebuild` is two sources of truth.

Collections are FODs (or nixpkgs packages), not `ansible-galaxy install` on Monday morning on the runner. Inventory lives in git. Ad-hoc `ansible all` is how a leftover gets a surprise reboot.

Plan the reinstall to NixOS. Ansible is a bridge with a retirement date.

Worked examples

Case 1: Pinned ansible CLI

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk Ansible (legacy only)";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      devShells.x86_64-linux.default = pkgs.mkShell {
```

```
        packages = [ pkgs.ansible ];
      };
    };
}
```

```
nix develop --command ansible --version
which ansible
```

`which` must be a store path. `~/.local/bin/ansible` is the unpinned CLI you just tried to retire.

Case 2: Inventory in git

Save as `inventory.ini`:

```
# inventory.ini
[legacy]
ubuntu-leftover.desk.internal

[nixos]
app-01.desk.internal
```

```
nix develop --command ansible legacy -i inventory.ini -m ping
```

Do not put NixOS hosts in the group you run playbooks against. The `[nixos]` group exists so Case 5 can refuse them.

Case 3: Collections pinned, not Galaxy at runtime

If nixpkgs has the collection, use it. If not, vendor the tarball as a fixed-output derivation and point `ANSIBLE_COLLECTIONS_PATH` at it.

Save as `collections.nix`:

```
# collections.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.stdenv.mkDerivation {
  pname = "desk-ansible-collections";
  version = "1.0.0";
  src = pkgs.fetchurl {
    url = "https://example.invalid/community-general-9.0.0.tar.gz";
    hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  };
  installPhase = ''
    mkdir -p $out/ansible_collections
```

```
    tar -C $out/ansible_collections -xzf $src
  '';
}
```

```
# in the devShell
{
  packages = [ pkgs.ansible collections ];
  ANSIBLE_COLLECTIONS_PATH = "${collections}";
}
```

Do not `ansible-galaxy install` on the runner every job. That is `terraform init` downloading providers again.

Case 4: Playbook that only talks to leftover Ubuntu

Save as `ping.yml`:

```
# ping.yml
- hosts: legacy
  gather_facts: false
  tasks:
    - ping:
```

```
nix develop --command ansible-playbook -i inventory.ini ping.yml
```

If this playbook grows `apt: name=nginx`, you are postponing the NixOS reinstall. Put a date in `README.md`.

Case 5: Refuse NixOS in the inventory

Save as `assert.yml`:

```
# assert.yml
- hosts: all
  gather_facts: true
  tasks:
    - name: refuse NixOS
      ansible.builtin.fail:
        msg: "Use nixos-rebuild / Colmena, not Ansible"
      when: ansible_facts['os_family'] | default('') == 'NixOS'
```

```
nix develop --command ansible-playbook -i inventory.ini assert.yml
```

A NixOS host in `all` must fail the play. That failure is the boring default working.

The trap

The trap is **Ansible `copy` of a `configuration.nix` then `shell: nixos-rebuild`**. You now have two sources of truth, and the one that wins is whoever ran last. SSH to the host and rebuild from the flake, or Colmena from `ops-01`.

The boring rule

- NixOS hosts: no Ansible. Fail the play if `os_family` is NixOS.
- Legacy hosts: Ansible from a Nix-pinned CLI + vendored collections.
- Inventory in git. No ad-hoc `ansible all`.
- Collections are FODs, not galaxy-on-Monday.
- Plan the reinstall to NixOS; Ansible is a bridge with a date.

Try this

1. `which ansible` inside `nix develop` — store path.
2. Add a fake inventory host, `ansible-inventory --list`.
3. `git grep ansible` on NixOS modules — should be empty.
4. Write the retirement date for the last Ubuntu leftover in the desk README.

Cloud Provider Configuration

Cloud Provider Configuration

Cloud consoles are mutable. The boring default is: **NixOS images from nixos-generators (or official AMIs), cloud-init only for SSH keys at first boot, then the flake owns the instance**.

Mental model

Piece	Owner
AMI / qcow / gce image	`nixos-generators` on 26.05, or official NixOS AMIs
Instance size, disk, VPC	OpenTofu `withPlugins`
OS config	`nixosConfigurations` in the same repo
First-boot SSH	cloud-init / metadata **once**

Do not bake AWS keys into the image. Do not grow `user_data` into a second `configuration.nix`. After first boot, Colmena / `nixos-rebuild --flake` takes over.

The NixOS firewall and the cloud security group must tell the same story. `0.0.0.0/0` on 22 "for debug" is the story of a scanner.

Worked examples

Case 1: Official AMI lookup (OpenTofu)

Save as `ami.tf`:

```
# ami.tf
data "aws_ami" "nixos" {
  owners      = ["427812963091"]
  most_recent = true
  filter {
    name   = "name"
    values = ["nixos/26.05*"]
  }
  filter {
    name   = "architecture"
    values = ["x86_64"]
```

```
  }
}
```

Owner ID is the NixOS AMI account as documented on nixos.org/download. Confirm it when you copy. `most_recent` plus a name glob is a moving target — pin a **specific AMI id** once you have tested it:

```
# ami-pin.tf
variable "nixos_ami" {
  type    = string
  default = "ami-xxxxxxxxxxxxxxxxx"
}
```

Case 2: Image you built

```
nix build .#amazon
ls -l result
# upload / AMI import per current AWS docs for the VHD
```

The generator produces the artifact. This chapter only consumes the AMI id in tofu. Do not hand-edit an AMI from 24.11 after the flake moved to 26.05.

Case 3: user-data is SSH keys, not packages

Save as `userdata.tf`:

```
# userdata.tf
resource "aws_instance" "app_01" {
  ami           = var.nixos_ami
  instance_type = "t3.medium"
  user_data     = <<-EOT
    #cloud-config
    users:
      - name: deskadmin
        sudo: ALL=(ALL) NOPASSWD:ALL
        ssh_authorized_keys:
          - ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI… deskadmin@ops-01
  EOT
  tags = {
    Name    = "desk-app-01"
    Flake   = "git.desk.internal/platform/desk"
    HostKey = "app-01"
  }
}
```

After first boot:

```
ssh deskadmin@203.0.113.11
sudo nixos-rebuild switch --flake git+ssh://git.desk.internal/platform/desk#app-01
```

Do not keep adding packages to `user_data`. They vanish on the next rebuild anyway if the flake does not declare them.

Case 4: Security group matches the firewall

Save as `sg.tf`:

```
# sg.tf
resource "aws_security_group" "gw" {
  name = "desk-gw"
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 51820
    to_port     = 51820
    protocol    = "udp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["203.0.113.0/24"] # office VPN, not the world
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

On the host, `networking.firewall.allowedTCPPorts = [ 443 ];` and `allowedUDPPorts = [ 51820 ];`. If the SG is wider than NixOS, the SG is the bug. If NixOS is wider than the SG, the module is the bug.

Case 5: Tags are how you find the box

When DNS is wrong:

```
aws ec2 describe-instances --filters Name=tag:HostKey,Values=app-01 \
  --query 'Reservations[].Instances[].InstanceId'
```

Tags are not secrets. Do not put the restic password in a tag.

The trap

The trap is **an AMI from 24.11 still in tofu** after the flake moved to 26.05. The instance boots old software, then the first rebuild is a surprise kernel. Pin AMI name `nixos/26.05*` **or** a specific AMI id you tested — and bump it on purpose.

The boring rule

- Images: 26.05 generator or official 26.05 AMI, then pin the id.
- tofu owns cloud objects. Flake owns the OS.
- user-data: SSH keys only. Then Colmena.
- SG + NixOS firewall agree. No `0.0.0.0/0` on 22.
- No cloud keys in the image.

Try this

1. `aws ec2 describe-images --owners 427812963091 --filters Name=name,Values=nixos/26.05*` (if you have AWS) and record an AMI id.
2. Compare that id to what tofu would pick with `most_recent`.
3. List ports in the SG vs `networking.firewall.allowedTCPPorts` for one host — they should match.
4. Boot from user-data keys, then rebuild from the flake; confirm a package you did **not** put in user-data is present only because the flake says so.

Multi-Cloud Deployment Patterns

Multi-Cloud Deployment Patterns

Two clouds is two IAM models, two images, two outages. The boring default is: **one cloud until you have a written reason; if you must span, one flake with two `nixosConfigurations` and two tofu roots — not a unified abstraction layer**.

Mental model

NixOS **closures** are cloud-agnostic. **Images and metadata** are not. An AWS AMI is not a GCP image. `nixos-generators` emits per-format artifacts from the **same** modules.

```
modules/desk-api.nix
      nixosConfigurations.desk-aws   → amazon image
      nixosConfigurations.desk-gcp   → gce image
tofu/aws/   state in s3://desk-tf-state/aws
tofu/gcp/   state in gcs://desk-tf-state/gcp
```

Do not share state files across clouds. Do not `for_each {aws, gcp}` in one root. A lock in the wrong cloud is an outage in the right one.

WireGuard between clouds is a mesh, not a reason to invent a "cloud-agnostic" Nix wrapper.

Worked examples

Case 1: Shared module, two hosts

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk two-cloud (only if you must)";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      mk = name: extra: nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
```

```
            ./modules/common.nix
            extra
            { networking.hostName = name; }
          ];
        };
    in
    {
      nixosConfigurations = {
        desk-aws = mk "desk-aws" ./hosts/aws.nix;
        desk-gcp = mk "desk-gcp" ./hosts/gcp.nix;
      };
    };
}
```

Save as `modules/common.nix`:

```
# modules/common.nix
{
  system.stateVersion = "26.05";
  services.openssh.enable = true;
  services.openssh.settings.PasswordAuthentication = false;
  networking.firewall.enable = true;
  # desk-api service lives here
}
```

`aws.nix` / `gcp.nix` own disks, metadata, and the provider's idea of a NIC. `common.nix` owns users, sshd, desk-api.

Case 2: Same birth, two clouds

`stateVersion` is birth, not channel. If both hosts are new on 26.05:

```
# already in common.nix
{ system.stateVersion = "26.05"; }
```

If `desk-aws` was born on 24.11, leave **that** host at `"24.11"` and only set `"26.05"` on the new GCP box. Copy-pasting `stateVersion` across clouds is how you skip a migration.

Case 3: Images from the generator, not a conversion script

```
nix build .#amazon
nix build .#gce
```

Typical artifacts (names follow your generator pin):

```
result/nixos-amazon-image-26.05.vhd
result/nixos-image-26.05-x86_64-linux.raw.tar.gz
```

Do not convert an AMI to GCE with a random script if the generator already has `gce` / `googleComputeImage`. Two formats, one module set.

Case 4: Separate tofu directories

```
tofu/aws/main.tf
tofu/aws/backend.tf    # bucket desk-tf-state, key desk/aws
tofu/gcp/main.tf
tofu/gcp/backend.tf    # bucket desk-tf-state, key desk/gcp
```

Save as `tofu/aws/backend.tf`:

```
# tofu/aws/backend.tf
terraform {
  backend "s3" {
    bucket = "desk-tf-state"
    key    = "desk/aws/terraform.tfstate"
    region = "eu-central-1"
  }
}
```

Different backends. Different credentials. Apply from `tofu/aws` never loads GCP state.

Case 5: WireGuard between clouds, not a service mesh

If the only reason for two clouds is "HA," a WireGuard mesh (`10.100.0.0/24`, `/32` peers) plus Colmena is enough. `desk-aws` is `10.100.0.11`, `desk-gcp` is `10.100.0.12`. UDP 51820 on each public NIC. Postgres still only on `wg0`.

Multi-cloud Kubernetes is not a first project. It is a second control plane, a second IAM, and a second way to lose DNS.

```
ls tofu/aws tofu/gcp
test ! -f tofu/terraform.tfstate && echo 'no shared state at tofu/ (good)'
```

If a single `terraform.tfstate` sits at `tofu/`, you already merged the roots. Split them before the next apply.

The trap

The trap is **a "cloud-agnostic" Nix wrapper** that hides AMI vs GCE differences until `tofu apply`. Keep the differences in `aws.nix` / `gcp.nix` where you can read them. The other trap is one terraform state that `for_each`s both clouds.

The boring rule

- One cloud first. Write the sentence that justifies the second.
- Same NixOS modules; different image formats and tofu roots.
- `stateVersion` per host birth — not "whatever the other cloud has."
- No shared terraform state.
- Mesh if you need routing; not a service mesh across clouds on day one.

Try this

1. List what actually differs between two clouds for the desk (image, IP, disk, IAM).
2. Draw two tofu dirs on paper. Do not merge them.
3. Build `.#amazon` and `.#gce` from one module set.
4. Write the one-sentence reason you need the second cloud — or delete the work.

Secrets and Security

Why Secrets Are Hard with Declarative Config

Why Secrets Are Hard with Declarative Config

Nix wants every input in the expression. Attackers want every input too. The boring default is: **treat `/nix/store` as public, never put secret bytes in Nix, and decrypt only onto `/run` at boot**.

Mental model

`/nix/store` items are world-readable. Binary caches replicate them. CI logs echo eval results. A private git repo does **not** make a string in `configuration.nix` private — the next `nix copy` to a cache, a coworker, or a backup disk copies the secret.

Safe in the flake	Never in the flake
Path `/run/secrets/db`	Password string
Age **public** keys	Age **private** keys
sops *file names*	Decrypted YAML
`hashedPasswordFile = "/persist/…"`	`initialPassword = "hunter2"`

Builds are sandboxed: they cannot prompt you. If a derivation "needs" the production DB password, the design is wrong. Fetch private **source** with `access-tokens`. Fetch private **runtime** secrets on the machine that runs the unit.

```
git  →  encrypted blob  →  eval knows the path
boot →  decrypt to tmpfs /run/secrets  →  unit EnvironmentFile
```

Worked examples

Case 1: Prove the leak

Save as `leak.nix`:

```
# leak.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.writeText "database.conf" ''
  password = SuperSecretPassword123
''
```

```
nix-build leak.nix
cat result
stat -c '%a %n' "$(readlink -f result)"
```

Output:

```
password = SuperSecretPassword123
444 /nix/store/…-database.conf
```

Mode `r--r--r--`. Any local uid can `cat` it. `nix copy --to ssh://untrusted` sends it off-box.

Case 2: `builtins.readFile` is the same leak

Save as `read.nix`:

```
# read.nix
builtins.readFile ./prod.password
```

```
echo 'hunter2' > prod.password
nix-instantiate --eval read.nix
```

Output:

```
"hunter2\n"
```

The string is now in the eval. If a module interpolates it into a unit, it is in the store. Use `readFile` only with **public** files (a banner, a JSON schema).

Case 3: Reference a runtime path

Save as `safe.nix`:

```
# safe.nix
{ config, pkgs, ... }:

{
  systemd.services.desk-api = {
    wantedBy = [ "multi-user.target" ];
    serviceConfig = {
      ExecStart = "${pkgs.hello}/bin/hello";
      EnvironmentFile = "/run/secrets/desk-api.env";
      DynamicUser = true;
    };
  };
}
```

Eval contains the **path**. The file is created at boot by sops-nix / agenix / Vault agent. Check:

```
nix why-depends /run/current-system /nix/store/…-database.conf || echo 'not in the OS closur
```

`ExecStart=… --token hunter2` is the same leak as Case 1. `EnvironmentFile` / `LoadCredential` are the boring APIs.

Case 4: Hashes versus plaintext

`hashedPassword = "$6$…"` is still in the store. It is an **offline crack** target, not a live password. Prefer `hashedPasswordFile` on a `0400` persist path. Never `initialPassword`. Same for `users.users.root.hashedPassword`.

```
# hash-ok-ish.nix
{
  users.users.deskadmin.hashedPasswordFile = "/persist/secrets/deskadmin.hash";
}
```

```
sudo install -m 0400 /dev/null /persist/secrets/deskadmin.hash
# write the mkpasswd hash there; not into git
```

Case 5: Hunt the closure

```
# after a switch, on a lab VM - dummy strings only
sudo grep -R "SuperSecret" /nix/store 2>/dev/null | head
```

If grep hits, that generation is tainted. Roll back, rotate, GC, and if you pushed a cache, consider the secret public. `nix flake check` must not need production secret **values**.

```
nix why-depends /run/current-system /nix/store/…-database.conf
```

A hit means the OS closure still **references** the leak. Deleting the file from git is not enough until a new generation GC's that path — and every cache that already copied it.

The trap

The trap is **private GitHub = safe**. Clones, Actions artifacts, `nix copy` to the team cache, and a laptop left at security all see the store. Encrypt in git (sops/age) so the repo is safe to copy. Decrypt on the host.

The other trap is putting the sops **private** key in the flake so CI can decrypt. Then CI logs are the leak. CI uses a runner identity; laptops use laptop keys.

A third: `systemd.services.*.environment.DB_PASSWORD = "…"` or `extraConfig` interpolating `builtins.readFile`. `LoadCredential` / `EnvironmentFile` of a **path** is the API.

The boring rule

- Store is public. Write configs as if a stranger can `cat` every NAR.
- Paths in Nix. Bytes on `/run` (tmpfs) or a `0400` persist file.
- No `readFile` of secrets. No secret strings in `environment` or argv.
- After a leak: rotate, then assume copies exist (cache, clones, backups).
- `nix flake check` must not need production secret **values**.

Try this

1. Case 1, then `stat -c '%a %n' $(readlink -f result)` — expect `444`.
2. Put a dummy token in a `.nix` file, `nixos-rebuild dry-build`, `grep` the built unit in the store. Remove it.
3. `git grep -nE 'initialPassword|privateKey =|password =' -- '*.nix'` on the desk repo.
4. Confirm `/run/secrets` is tmpfs (`findmnt /run`) on a sops-enabled host.

Using SOPS for Encrypted Configuration (sops-nix)

Using SOPS for Encrypted Configuration (sops-nix)

Secrets that live in a wiki are missing on the next host. The boring default is: **SOPS YAML in git (keys visible, values encrypted) and `sops-nix` decrypting to `/run/secrets` at boot**.

Mental model

Mozilla SOPS encrypts **values** in YAML/JSON. Key names stay plaintext so diffs stay reviewable (`database/password:` still reads as that path). Recipients are Age keys (or SSH host keys converted with `ssh-to-age`).

`sops-nix` is a NixOS module:

1. Reads `defaultSopsFile`.
2. At activation/boot, decrypts with the host Age key (`age.keyFile` or SSH host key).
3. Writes files under `/run/secrets/…` with `owner` / `mode` you set.
4. Units reference `config.sops.secrets.<name>.path`.

```
secrets/desk.yaml   (cipher in git)
          sops-nix + host age key

/run/secrets/database/password   (0400, tmpfs)


desk-api EnvironmentFile=
```

Worked examples

Case 1: Recipients and a secrets file

Save as `.sops.yaml`:

```yaml
# .sops.yaml
creation_rules:
  - path_regex: secrets/.*\.yaml$
    key_groups:
      - age:
          - age1deskadminxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
          - age1desknode01xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
mkdir -p secrets
sops secrets/desk.yaml
```

Example **decrypted** editor view (what sops shows you):

```
database:
  password: please-change-me
```

On disk in git, the value is `ENC[AES256_GCM,data:…]`. Commit both `.sops.yaml` and `secrets/desk.yaml`.

Case 2: NixOS module

Save as `secrets-module.nix`:

```
# secrets-module.nix
{ config, inputs, ... }:

{
  imports = [ inputs.sops-nix.nixosModules.sops ];

  sops = {
    defaultSopsFile = ./secrets/desk.yaml;
    defaultSopsFormat = "yaml";
    age.keyFile = "/var/lib/sops-nix/key.txt";

    secrets."database/password" = {
      owner = "desk-api";
      mode = "0400";
    };
  };

  systemd.services.desk-api.serviceConfig.EnvironmentFile =
    config.sops.secrets."database/password".path;
}
```

If the YAML value is a raw password, `EnvironmentFile` is the wrong shape (that file is `KEY=value` lines). Store a dotenv in sops (`DESK_DB=…`) or use a **template**:

```
# sops-template.nix fragment
{
  sops.templates."desk-api.env" = {
    content = ''
      DATABASE_PASSWORD=${config.sops.placeholder."database/password"}
```

```
    '';
    owner = "desk-api";
    mode = "0400";
  };
  systemd.services.desk-api.serviceConfig.EnvironmentFile =
    config.sops.templates."desk-api.env".path;
}
```

`placeholder` is ciphertext-safe at eval: Nix never sees the live password. `restartUnits = [ "desk-api.service" ];` on the secret so a rekey + switch restarts the unit.

```
nix eval --raw .#nixosConfigurations.desk-server.config.sops.secrets.\"database/password\".p
```

Output:

```
/run/secrets/database/password
```

Case 3: Age key from SSH host key

```
# sops-ssh.nix
{
  sops.age.sshKeyPaths = [ "/etc/ssh/ssh_host_ed25519_key" ];
}
```

No extra key to distribute if the host already has an Ed25519 host key **and** that key is in `.sops.yaml` (via `ssh-to-age`). Persist the host key under impermanence or every reboot is a new recipient.

```
ssh-to-age < /etc/ssh/ssh_host_ed25519_key.pub
```

Paste the `age1…` line into `.sops.yaml`, then `sops updatekeys secrets/desk.yaml`.

Case 4: Flake input

```
# flake.nix fragment
{
  inputs.sops-nix = {
    url = "github:Mic92/sops-nix";
    inputs.nixpkgs.follows = "nixpkgs";
  };
}
```

`follows` so sops-nix does not pull a second nixpkgs.

Case 5: Missing owner

If `desk-api` runs as `DynamicUser` and the secret is `0400 root`, the unit fails with `Permission denied`. Set `owner` to the service user, or use `sops.secrets.x.restartUnits = [ "desk-api.service" ];` plus a group both can share.

```
sudo ls -l /run/secrets/database/password
sudo systemctl status desk-api
```

The trap

The trap is **encrypting only to your laptop Age key**. The server cannot decrypt at boot. Always include the **host** recipient (and the admin laptop, and CI if CI must eval decrypt).

The other trap is `age.keyFile` on a path that impermanence wipes. Persist `/var/lib/sops-nix` or use the SSH host key that you already persist.

The boring rule

- Encrypted YAML in git. Recipients: admins + each host.
- Units consume `.path` or `sops.templates`. Never decrypted strings in Nix. CI `flake check` must not need the private key.
- Host Age key = SSH host key when you can. Persist that key.
- Set `owner` and `mode` to match the unit.
- `sops updatekeys` when a machine or person is added or revoked.

Try this

1. `age-keygen -o /tmp/k.txt`, add the public key to `.sops.yaml`, `sops secrets/demo.yaml` with `hello: world`, `sops -d secrets/demo.yaml`.
2. `ssh-to-age < ~/.ssh/id_ed25519.pub` and encrypt a file to that recipient.
3. Eval `.path` from a flake that imports sops-nix. Confirm it starts with `/run/secrets`.
4. Intentionally omit the host from `.sops.yaml`, rebuild a VM, read the sops-nix unit log. Add the host and `updatekeys`.

HashiCorp Vault Integration

HashiCorp Vault Integration

sops-nix and age cover **static** secrets in git. Large desks that already run Vault want **short-lived** database passwords and certs. The boring default is: **Vault Agent on the node, templates to `/run/vault`, Nix only knows the paths — and if you do not already operate Vault, stay on sops-nix**.

Mental model

Vault issues leases. The agent:

1. Authenticates (AppRole, AWS, Kubernetes — pick one).
2. Reads secrets / PKI.
3. Renders a file on tmpfs.
4. Signals the unit (`SIGHUP` or restart) when the file changes.

NixOS may ship `services.vault-agent` — **check your 26.05 pin** (`nixos-option services.vault-agent`). If the option is missing, Case 1's `systemd.services.vault-agent-desk` is the boring wrapper. The **role-id** can live in the store. The **secret-id** cannot. Vault Agent is a **node sidecar**, not a reason to put `VAULT_TOKEN` in `desk-api`'s Nix module.

```
Vault  --lease-->  vault-agent  --template-->  /run/vault/db.env


                                                desk-api EnvironmentFile=
```

Worked examples

Case 1: Agent module

Save as `vault-agent.nix`:

```nix
# vault-agent.nix
{ pkgs, ... }:

{
  systemd.services.vault-agent-desk = {
    description = "Vault Agent for desk-api";
    wantedBy = [ "multi-user.target" ];
```

```
    after = [ "network-online.target" ];
    serviceConfig = {
      ExecStart = "${pkgs.vault}/bin/vault agent -config /etc/vault-agent.hcl";
      Restart = "on-failure";
      DynamicUser = true;
      ProtectSystem = "strict";
      PrivateTmp = true;
      RuntimeDirectory = "vault-agent";
      LoadCredential = "secret-id:/run/secrets/vault-secret-id";
    };
  };

  environment.etc."vault-agent.hcl".text = ''
    vault { address = "https://vault.desk.internal:8200" }
    auto_auth {
      method "approle" {
        config = {
          role_id_file_path = "/etc/vault/role-id"
          secret_id_file_path = "/run/secrets/vault-secret-id"
        }
      }
    }
    template {
      source      = "/etc/vault/templates/db.ctmpl"
      destination = "/run/vault-agent/db.env"
      perms       = "0400"
    }
  '';
}
```

`role-id` is not a secret in many setups. `secret-id` is — put it in sops-nix, not in `environment.etc`.

Case 2: Template without secret values in Nix

Save as `db.ctmpl` (provisioned via `environment.etc` — this file has **Go template** placeholders, not passwords):

```
# db.ctmpl
{{ with secret "database/creds/desk-api" }}
DESK_DB_USER={{ .Data.username }}
DESK_DB_PASSWORD={{ .Data.password }}
{{ end }}
```

The rendered `/run/vault-agent/db.env` never enters `/nix/store`.

Case 3: Reload the app when the lease rotates

```
# vault-reload.nix
{
  systemd.services.desk-api = {
    serviceConfig.EnvironmentFile = "/run/vault-agent/db.env";
    unitConfig.StartLimitIntervalSec = 0;
  };

  systemd.paths.desk-api-reload = {
    wantedBy = [ "multi-user.target" ];
    pathConfig.PathModified = "/run/vault-agent/db.env";
  };

  systemd.services.desk-api-reload = {
    serviceConfig = {
      Type = "oneshot";
      ExecStart = "/run/current-system/sw/bin/systemctl reload-or-restart desk-api.service";
    };
  };
}
```

If the app cannot reload credentials without a restart, restart. Document that. Leases that expire while the process still holds the old password are outages.

Case 4: AppRole files

```
# vault-approle.nix
{ config, ... }:

{
  environment.etc."vault/role-id".text = "00000000-0000-0000-0000-000000000000";

  # secret-id from sops, not from etc
  sops.secrets."vault/secret-id" = {
    mode = "0400";
  };
}
```

Use a real UUID from `vault read auth/approle/role/desk-api/role-id`. The example zeros are a placeholder so this listing does not pretend to be a live role.

Case 5: When not to use Vault

If the desk has five hosts and two humans, Vault is a second production system. sops-nix + age is the boring stack. Add Vault when the org already requires it, or when credentials **must** rotate

every hour.

The trap

The trap is **a root Vault token in `configuration.nix`**. That token is in the store and in every cache copy. AppRole + a sops-backed secret-id. Never `VAULT_TOKEN` in the flake.

The other trap is ignoring lease TTL. Agent templates that nobody reloads leave the DB password valid in Vault's eyes for twenty minutes after the process started — then Postgres rejects it at 3 a.m.

The boring rule

- Nix knows addresses, role ids, and **paths**. Bytes stay in Vault and `/run`.
- AppRole (or cloud auth). No root tokens in git.
- Reload or restart on template change.
- secret-id via sops-nix. role-id may be public.
- Skip Vault if sops-nix already solves the desk.

Try this

1. `nixos-option services.vault-agent` (or `rg vault-agent` in nixpkgs) on a 26.05 pin and write down the module path your release actually ships.
2. Render a dummy `db.ctmpl` with static `{{ printf "x" }}` locally using `consul-template` / vault agent `-dev` if you have a lab Vault. Confirm the output file is not a store path.
3. `git grep VAULT_TOKEN` on the desk repo. It should be empty.
4. Time a database lease TTL and confirm desk-api restarts before expiry in the lab.

Age Encryption for Simple Secrets

Age Encryption for Simple Secrets

GPG keyrings and expiry theatre do not belong in a desk CI job. The boring default is: **age (X25519) for files, `ssh-to-age` so SSH keys are recipients, and `agenix` only if you want NixOS decryption without SOPS's YAML**.

This book already uses **sops-nix** as the default for structured secrets. Age is the primitive underneath. Agenix is the alternative when you want one file per secret and less YAML.

Mental model

Tool	Role
`age`	Encrypt/decrypt bytes to `age1…` or SSH recipients
`ssh-to-age`	Convert `ssh-ed25519` pubkey → `age1…`
`agenix`	NixOS module: `.age` files in git → `/run/agenix/…` at boot

No keyserver. A public key is a line of text. Lose the private key, lose the files (unless another recipient remains).

```
desk-token.age   (cipher, git)
   recipients: alice's ssh, desk-node host ssh
           agenix at boot

/run/agenix/desk-token
```

Private keys never enter `/nix/store`. Ciphertext always lives in git.

```
git check-ignore -v key.txt .age/key.txt 2>/dev/null || true
git ls-files '*.age' | head
```

`.age` files are tracked. `key.txt` is not. If `git ls-files` shows `key.txt`, rotate that key now.

Worked examples

Case 1: Encrypt a file to yourself

```
mkdir -p "$HOME/.age"
age-keygen -o "$HOME/.age/key.txt"
# public key printed on stderr: age1…
echo 'desk-token-v1' | age -r age1youxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx > token.age
age -d -i "$HOME/.age/key.txt" token.age
```

Output:

```
desk-token-v1
```

Commit `token.age`. Do not commit `key.txt`. `chmod 600` the key file.

`SOPS_AGE_KEY_FILE=$HOME/.age/key.txt` is how sops finds the same identity. `age -d -i` is the explicit form. Do not `export SOPS_AGE_KEY` (the raw key) in a shell history. Armor (`age -a`) is optional; sops YAML already stores ciphertext as ASCII.

Case 2: SSH public key as recipient

```
ssh-to-age < ~/.ssh/id_ed25519.pub
echo 'desk-token-v1' | age -R ~/.ssh/id_ed25519.pub -o token.age
age -d -i ~/.ssh/id_ed25519 token.age
```

Hosts:

```
ssh-to-age < /etc/ssh/ssh_host_ed25519_key.pub
```

Every NixOS machine already has a recipient if you persist host keys (impermanence: persist `/etc/ssh/ssh_host_ed25519_key` or decryption fails every boot).

`age -R recipients.txt` reads many `age1…` / `ssh-ed25519` lines. Keep `recipients.txt` in git (public keys only). A passphrase (`age -p`) is a human backup, not a fleet identity — sops/agenix will not type it at boot.

Case 3: agenix rules file

Save as `secrets.nix`:

```
# secrets.nix
let
  alice = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleUserKey alice@desk";
  server = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIExampleHostKey root@desk-node";
in
```

```
{
  "desk-token.age".publicKeys = [ alice server ];
}
```

```
nix shell nixpkgs#agenix --command agenix -e desk-token.age
```

`$EDITOR` opens the plaintext. Save; the `.age` file is updated. Recipients are **only** those `publicKeys`. Adding a host later requires `agenix -r` (rekey).

Case 4: Consume on NixOS

Save as `host.nix`:

```
# host.nix
{ config, inputs, ... }:

{
  imports = [ inputs.agenix.nixosModules.default ];

  age.secrets.desk-token = {
    file = ./desk-token.age;
    owner = "desk-api";
    group = "desk-api";
    mode = "0400";
  };

  systemd.services.desk-api.serviceConfig.LoadCredential =
    "token:${config.age.secrets.desk-token.path}";
}
```

Boot decrypts to `/run/agenix/desk-token` (path may be the credential copy). The unit never sees a Nix string. `EnvironmentFile` pointing at the secret path is also fine; argv is not (`ExecStart=… --token hunter2`).

`age.identityPaths` defaults to the host ed25519 key. If you also decrypt as a user, list that user's key — and do not put it in the store. `owner = "desk-api"` requires that user to **exist at decrypt time**; with `DynamicUser`, keep the file root-owned `0400` and `LoadCredential`.

Case 5: Rekey when the fleet grows

```
# add bob to secrets.nix publicKeys, then:
agenix -r
git diff desk-token.age
```

The ciphertext changes; plaintext is the same. If you skip rekey, bob's laptop cannot decrypt, and a new server fails at boot with an age error in the agenix unit.

```
journalctl -u agenix -n 40 --no-pager
```

The trap

The trap is **one recipient: your laptop**. The server is not you. Always include the host SSH key.

The other trap is using **both** sops-nix and agenix for the same secret. Pick one tree: sops YAML for many keys, agenix for a handful of files. Two systems means two rekey rituals.

A third: putting `key.txt` in the ISO or in `isoImage.contents`. Ciphertext in git; identity keys on disk / sops.

A fourth: `ssh-to-age` of a **private** key (it wants the `.pub`). A fifth: RSA host keys — prefer ed25519; `ssh-to-age` wants the modern key type.

The boring rule

- Age, not GPG, for desk secrets.
- Recipients = people + hosts (`ssh-to-age` of **`.pub`**). Persist host keys on ephemeral roots.
- Rekey when keys are added or revoked (`agenix -r` / `sops updatekeys`).
- Private keys never in git (`SOPS_AGE_KEY_FILE`, not `SOPS_AGE_KEY` in history). Ciphertext always in git.
- Prefer sops-nix if you already have YAML; agenix if you want one-file-one-secret. Not both for the same token.

Try this

1. `ssh-to-age < ~/.ssh/id_ed25519.pub`, encrypt a one-line file, decrypt with `-i ~/.ssh/id_ed25519`.
2. Encrypt to two recipients, remove your key from the list, confirm decrypt fails, put it back.
3. `agenix -e` a dummy secret in a throwaway repo and `git add` only the `.age` file.
4. On a VM, omit the host from `publicKeys`, switch, read `journalctl -u agenix`. Add the host, rekey, switch again.

Security Hardening Patterns

Security Hardening Patterns

Open ports, writable `/`, and JIT-friendly sysctls are the defaults of a general-purpose distro. The boring default on a desk **server** is: **firewall on, `DynamicUser` + `ProtectSystem=strict` on custom units, a small sysctl set, and no hardening that you have not booted**.

Mental model

Defense in this book is already mostly done if you followed earlier chapters: immutable store, listed firewall ports, no plaintext secrets, SSH keys only. This chapter is the extra kernel/unit knobs.

Layer	Boring control
Network	`networking.firewall.enable = true;`
Units	`ProtectSystem`, `ProtectHome`, `NoNewPrivileges`, `DynamicUser`
Kernel	`boot.kernel.sysctl` for kptr, bpf, syncookies
Firmware	microcode (hardware chapter)
Audit	`systemd-analyze security <unit>`

`MemoryDenyWriteExecute = true` breaks Node, JVM, and some Python. Test, then enable. A unit that will not start is not hardened; it is down.

NixOS 26.05's firewall is **nftables**. `networking.firewall.enable = true` is enough; do not also enable `iptables` "for the gist." `ip_forward` only on the WireGuard gateway.

Do not paste a 40-line gist onto `systemd.services.nginx` until `systemctl cat nginx` shows you still need those flags. Fighting the module produces restart loops. `linuxPackages_hardened` is **not** the desk default — firmware, ZFS, and vendor modules break. Measure, then maybe.

Worked examples

Case 1: Sandbox a desk unit

Save as `secure-service.nix`:

```
# secure-service.nix
{ pkgs, ... }:

{
```

```
  systemd.services.desk-api = {
    description = "Desk API";
    after = [ "network.target" ];
    wantedBy = [ "multi-user.target" ];
    serviceConfig = {
      ExecStart = "${pkgs.hello}/bin/hello";
      DynamicUser = true;
      ProtectSystem = "strict";
      ProtectHome = true;
      PrivateTmp = true;
      PrivateDevices = true;
      ProtectKernelTunables = true;
      ProtectKernelModules = true;
      ProtectControlGroups = true;
      RestrictRealtime = true;
      RestrictSUIDSGID = true;
      NoNewPrivileges = true;
      LockPersonality = true;
      SystemCallArchitectures = "native";
      RestrictAddressFamilies = [ "AF_INET" "AF_INET6" "AF_UNIX" ];
      CapabilityBoundingSet = "";
      AmbientCapabilities = "";
      UMask = "0077";
    };
  };
}
```

```
sudo nixos-rebuild switch
systemd-analyze security desk-api --no-pager | head -30
```

The score is a hint, not a compliance certificate. Read the **exposed** lines and decide.

Case 2: Kernel sysctl baseline

Save as `hardening.nix`:

```
# hardening.nix
{
  networking.firewall.enable = true; # nftables on 26.05

  boot.kernel.sysctl = {
    "kernel.unprivileged_bpf_disabled" = 1;
    "kernel.kptr_restrict" = 2;
    "kernel.yama.ptrace_scope" = 1;
    "net.ipv4.tcp_syncookies" = 1;
    "net.ipv4.conf.all.rp_filter" = 1;
```

```
    "net.ipv4.ip_forward" = 0;
  };
}
```

Turn `ip_forward` **on** only on the WireGuard gateway. A worker node that forwards is a mistake.

```
sysctl net.ipv4.ip_forward
```

`0` on `app-01`. `1` on `gw-01` only, and listed in the threat model.

Case 3: nixpkgs service modules already sandbox

```
# nginx-harden.nix
{
  services.nginx.enable = true;
  services.nginx.virtualHosts."desk.internal".locations."/".return = "204";
  networking.firewall.allowedTCPPorts = [ 80 ];
}
```

```
systemctl cat nginx | grep -E 'Protect|Private|NoNew'
```

Do not fight those with `mkForce` until you have a journal that says why.

Case 4: JIT exception

```
# node-api.nix
{
  systemd.services.desk-node.serviceConfig = {
    DynamicUser = true;
    ProtectSystem = "strict";
    # Node needs a writable executable mapping:
    MemoryDenyWriteExecute = false;
  };
}
```

Document the exception next to the option. A future reader will otherwise “fix” it and page themselves.

Case 5: Analyze before and after

```
systemd-analyze security sshd --no-pager | tail -5
systemd-analyze security desk-api --no-pager | tail -5
```

sshd will never look like a `DynamicUser` hello binary. Compare a unit to **itself** after a change, not to sshd.

The trap

The trap is **copying a 40-line hardening gist onto every unit**, including ones that need `/home`, BPF, or devices. The activation succeeds; the service dies; you disable the firewall to "debug." Add flags one at a time. Keep the firewall on.

The other trap is `MemoryDenyWriteExecute` on Go with CGO off (usually fine) copied blindly onto Node (usually not).

A third: `RestrictAddressFamilies` without `AF_UNIX` so the unit cannot talk to the journal / D-Bus and looks "broken." A fourth: `linuxPackages_hardened` on a box that needs NVIDIA or ZFS.

The boring rule

- Firewall (nftables), keys, secrets, and an immutable store first. Sysctls second.
- `DynamicUser` + `ProtectSystem=strict` + `RestrictAddressFamilies` on *your* network daemons.
- `MemoryDenyWriteExecute` only after the app starts under it.
- `ip_forward` only on routers. Hardened kernel is optional, not default.
- `systemd-analyze security` as a diff tool, not a scoreboard. Do not `mkForce` module sandboxes.

Try this

1. `systemd-analyze security desk-api` before and after Case 1. Note which checks flipped.
2. Set `MemoryDenyWriteExecute = true` on a Node unit in a VM, start it, read the journal, set it false.
3. `sysctl net.ipv4.ip_forward` on a worker vs the gateway.
4. `systemctl cat nginx` on 26.05 and list sandbox flags the module already set.

Platform Engineering

The Internal Developer Platform Pattern

The Internal Developer Platform Pattern

Fifty repos each pinning their own nixpkgs is fifty llvm builds. The boring default is: **one platform flake on nixpkgs 26.05 that exports `devShells`, `templates`, and `nixosModules`, and product flakes `follows` it**.

Mental model

```
git.desk.internal/platform     nixpkgs 26.05
       devShells.backend / frontend
       templates.go-service
       nixosModules.baseline
git.desk.internal/desk-api
    inputs.platform.url = …
    inputs.nixpkgs.follows = "platform/nixpkgs"
```

Upgrading Go or sshd defaults is a platform bump, then a lock bump in each consumer (or a bot). Not a meeting. Tag `platform-26.05.3`; consumers pin the tag, not `main`. A broken main at 09:01 must not break every laptop.

Keep the platform **thin**: shells, templates, twenty lines of sshd. An overlay that replaces half of nixpkgs is a fork you cannot debug.

Worked examples

Case 1: Platform flake

Save as `flake.nix` in the platform repo:

```
# flake.nix
{
  description = "Desk platform";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      systems = [ "x86_64-linux" "aarch64-linux" "aarch64-darwin" ];
      forAll = f: nixpkgs.lib.genAttrs systems (s: f nixpkgs.legacyPackages.${s});
```

```
    in
    {
      devShells = forAll (pkgs: {
        backend = pkgs.mkShell {
          packages = [ pkgs.go pkgs.gopls pkgs.postgresql ];
        };
        frontend = pkgs.mkShell {
          packages = [ pkgs.nodejs pkgs.pnpm ];
        };
      });

      templates.go-service = {
        path = ./templates/go-service;
        description = "Desk Go service";
      };

      nixosModules.baseline = ./modules/baseline.nix;
    };
}
```

Case 2: Init from template

Save as `templates/go-service/flake.nix` so `init` already `follows`:

```
# templates/go-service/flake.nix
{
  description = "Desk Go service";

  inputs.platform.url = "git+ssh://git.desk.internal/platform.git?ref=refs/tags/platform-26.
  inputs.nixpkgs.follows = "platform/nixpkgs";

  outputs = { self, nixpkgs, platform }:
    let
      pkgs = nixpkgs.legacyPackages.x86_64-linux;
    in
    {
      devShells.x86_64-linux.default = platform.devShells.x86_64-linux.backend;
      packages.x86_64-linux.default = pkgs.hello;
    };
}
```

```
nix flake init -t git+ssh://git.desk.internal/platform.git#go-service
```

Case 3: Consumer lock bump

Save as the product `flake.nix` (desk-api):

```
# flake.nix
{
  description = "desk-api";

  inputs.platform.url = "git+ssh://git.desk.internal/platform.git?ref=refs/tags/platform-26.
  inputs.nixpkgs.follows = "platform/nixpkgs";

  outputs = { self, nixpkgs, platform }: {
    devShells.x86_64-linux.default =
      platform.devShells.x86_64-linux.backend;
    nixosModules.default = { imports = [ platform.nixosModules.baseline ]; };
  };
}
```

```
nix flake update platform
nix flake metadata
```

`nix flake metadata` must show **one** nixpkgs. Two nixpkgs means a `follows` is missing.

Do not `nix flake update` of everything on Friday. Update `platform`, then only what broke.

Case 4: Baseline NixOS module

Save as `modules/baseline.nix`:

```
# modules/baseline.nix
{ lib, ... }:
{
  services.openssh.enable = lib.mkDefault true;
  services.openssh.settings.PasswordAuthentication = false;
  networking.firewall.enable = true;
  system.stateVersion = lib.mkDefault "26.05";
}
```

Hosts import `platform.nixosModules.baseline`. Hosts may `mkForce` in an emergency; review that in the PR. `mkDefault` is how a host opts into a different `stateVersion` if it was born earlier — birth still wins.

Case 5: Version the platform

```
git tag -a platform-26.05.3 -m "26.05.3 shells + sshd"
git push origin platform-26.05.3
```

Consumers pin the tag in `inputs.platform.url`. Floating `main` is for the platform team's laptops, not for production `desk-api`.

The trap

The trap is **a platform that overlays half of nixpkgs**. Consumers cannot debug `go` anymore: is it nixpkgs, the overlay, or the product? Keep the platform thin. One overlay for *one* pin, documented in `README.md`.

The boring rule

- One nixpkgs: 26.05 via the platform flake.
- Templates + `follows`. `nix flake metadata` shows a single nixpkgs.
- `nix flake update platform` is the upgrade API.
- Tags, not floating main, for production consumers.
- Thin modules. No mega-overlay.

Try this

1. `nix flake init -t` from a local path template.
2. `nix flake metadata` on a consumer and confirm a single nixpkgs.
3. Add a `devShells.docs` with `pkgs.pandoc` on the platform; consume it from desk-api.
4. Tag a fake `platform-26.05.3` and pin it in a consumer lock.

Monorepo Strategies with Nix

Monorepo Strategies with Nix

A monorepo with `src = ./.;` on every package rebuilds the world per commit. The boring default is: **one flake, many `packages.*`, `lib.fileset` (or `cleanSourceWith`) per leaf, and CI that builds `.#checks` — not `.#everything` — on a docs-only PR**.

Mental model

Layout	Nix
`apps/desk-api`	`packages.desk-api` with `src` filtered to that tree
`apps/desk-web`	sibling package; a docs change does not rebuild api
`modules/`	`nixosModules`
`flake.nix`	one lock, nixpkgs 26.05

`nix flake check` should be cheap enough to run on every PR. Heavy images are extra jobs or nightly. Recursive flakes (`inputs.foo.url = "path:./apps/foo"`) multiply locks — one lock at the root.

Nix caching still helps if you rebuild the world, but eval time and `vendorHash` churn do not. Filter first.

Worked examples

Case 1: fileset `src` per leaf

Save as `pkgs/desk-api.nix`:

```
# pkgs/desk-api.nix
{ pkgs, lib }:

pkgs.buildGoModule {
  pname = "desk-api";
  version = "1.0.0";
  src = lib.fileset.toSource {
    root = ../.;
    fileset = lib.fileset.unions [
```

```
      ../apps/desk-api
      ../go.mod
      ../go.sum
    ];
  };
  vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

A README typo under `apps/desk-web` is not in this fileset. `nix build .#desk-api` should be a cache hit.

Case 2: One flake, many packages

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk monorepo";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      packages.${system} = {
        desk-api = pkgs.callPackage ./pkgs/desk-api.nix { };
        desk-web = pkgs.callPackage ./pkgs/desk-web.nix { };
        default = self.packages.${system}.desk-api;
      };
      checks.${system} = {
        api = self.packages.${system}.desk-api;
        web = self.packages.${system}.desk-web;
      };
    };
}
```

```
nix build .#desk-api
nix flake check -L
```

Case 3: PR path filter (CI)

Save as `.github/workflows/leaf.yml`:

```yaml
# .github/workflows/leaf.yml
name: leaf
on:
  pull_request:
    paths:
      - "apps/desk-api/**"
      - "pkgs/desk-api.nix"
      - "go.mod"
      - "go.sum"
jobs:
  api:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: nix build .#desk-api
```

A second workflow watches `flake.lock` and runs `nix flake check`. Nix will still cache; the path filter saves eval time on huge repos.

Case 4: `linkFarm` for the nightly smoke set

```nix
# in flake outputs
{
  packages.x86_64-linux.desk-smoke = pkgs.linkFarm "desk-smoke" [
    { name = "api"; path = self.packages.x86_64-linux.desk-api; }
    { name = "web"; path = self.packages.x86_64-linux.desk-web; }
  ];
}
```

```
nix build .#desk-smoke
```

Nightly builds the farm. PRs build the leaf. Do not make `default` the farm if humans run `nix build` on every save.

Case 5: Do not use recursive flakes

```nix
# do not
{
  inputs.desk-api.url = "path:./apps/desk-api";
  inputs.desk-web.url = "path:./apps/desk-web";
}
```

Each path flake wants its own `flake.lock`. You will update nixpkgs in one leaf and not the other. `callPackage` plus filesets is the boring layout.

The trap

The trap is `src = ./.;` at repo root for every `callPackage`. A README typo rebuilds Go, invalidates `vendorHash` work, and CI is red for twenty minutes. Filter. The other trap is a recursive flake per app.

The boring rule

- One flake, one 26.05 lock.
- Per-package `lib.fileset` (or `cleanSourceWith`) `src`.
- Leaf builds on leaf PRs; `flake.lock` PRs run full check.
- No recursive flakes.
- `vendorHash` / `cargoHash` per package, not one hash for the repo.

Try this

1. `nix build .#desk-api`, then touch a web-only file; `nix build .#desk-api --dry-run` (should be empty if filtered).
2. Pick `lib.fileset` or `cleanSource` — one style per repo.
3. Add a `checks` attr that is `linkFarm` of two packages; run it only nightly.
4. Time `nix flake show` on the monorepo; if it is minutes, you eval too much at the top.

Multi-Platform Development and Distribution

Multi-Platform Development and Distribution

`x86_64-linux` CI does not prove `aarch64-darwin`. The boring default is: **`lib.genAttrs` over the systems humans use, native remote builders for foreign `system`, and no QEMU-on-the-laptop as the production path**.

Mental model

What	How
Same shell on Mac + Linux	`devShells.${system}` from nixpkgs 26.05
Linux binary from a Mac	Remote Linux builder (`ssh-ng`)
Raspberry Pi image	`aarch64-linux` builder or `nixos-generators` `sd-aarch64`

`pkgsCross` is **cross**. A remote builder is **native**. Native is faster and hits the team cache. Cross is for when you do not have the hardware (Go `GOOS`/`GOARCH` is a third path — still not QEMU).

```
each system  →  packages.${system}.desk-api
             →  devShells.${system}.default
foreign linux  →  --max-jobs 0  +  ssh-ng builder
```

Two archs are two cache objects. `nix copy` each one.

Worked examples

Case 1: Enumerate systems without flake-utils

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk multi-system";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
```

```
      systems = [ "x86_64-linux" "aarch64-linux" "aarch64-darwin" ];
      forAll = f: nixpkgs.lib.genAttrs systems (system: f {
        inherit system;
        pkgs = nixpkgs.legacyPackages.${system};
      });
  in
  {
    packages = forAll ({ pkgs, ... }: {
      default = pkgs.hello;
      desk-api = pkgs.hello; # replace with callPackage
    });
    devShells = forAll ({ pkgs, ... }: {
      default = pkgs.mkShell { packages = [ pkgs.go pkgs.gopls ]; };
    });
  };
}
```

`flake-utils` is optional sugar. One extra input is one extra `follows` to forget. `lib.genAttrs` is enough.

```
nix flake show
```

You should see three systems. Skip `x86_64-darwin` for new work: nixpkgs 26.11 drops it. 26.05 still builds it until EOL.

Case 2: Native remote builder, not qemu

On the Darwin laptop, do not emulate `aarch64-linux` as the release path:

```
nix build .#packages.aarch64-linux.desk-api --max-jobs 0
```

`--max-jobs 0` forbids local realisation. Nix must use `ssh-ng` to an aarch64 (or x86_64-linux) builder. If you have no builder:

```
error: a 'aarch64-linux' with features {} is required to build '…', but I am a 'aarch64-darwin'
```

That error is correct. Add a builder; do not reach for `qemu-binfmt` for the artefact you ship.

Save as `builders.nix` on the Darwin (or laptop) NixOS/nix-darwin host:

```
# builders.nix
{
  nix.buildMachines = [{
    hostName = "builder.desk.internal";
    system = "x86_64-linux";
    protocol = "ssh-ng";
    maxJobs = 8;
```

```
    speedFactor = 2;
    supportedFeatures = [ "kvm" "nixos-test" "big-parallel" ];
  }];
  nix.distributedBuilds = true;
  nix.settings.max-jobs = 0;
}
```

`supportedFeatures` must include `kvm` if you offload `runNixOSTest`. A missing flag is why tests compile on the laptop anyway.

Case 3: CI matrix for the OS humans use

```
# .github/workflows/build.yml
name: build
on: [pull_request]
jobs:
  build:
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest]
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v4
      - run: nix build .#packages.${{ matrix.os == 'macos-latest' && 'aarch64-darwin' || 'x8
```

Each job `nix build`. Cache per system. Do not emulate `aarch64-linux` on GHA ubuntu unless you accept 10× time and flaky KVM.

Case 4: Release two closures

```
nix build .#packages.x86_64-linux.desk-api -o result-x86
nix build .#packages.aarch64-linux.desk-api -o result-arm
nix path-info -Shr ./result-x86
nix path-info -Shr ./result-arm
nix copy --to ssh://cache.desk.internal ./result-x86 ./result-arm
```

Two closures. Two cache keys. Goreleaser is optional; `nix build` + `nix copy` is enough for internal desk tools.

Case 5: `file` the toolchain, not your hopes

On Apple silicon:

```
nix develop --command bash -c 'file "$(which go)"'
```

You want `arm64`, not `x86_64`. A hardcoded `system = "x86_64-linux"` in a flake used on a Mac may still eval; `go` is then the wrong ELF.

The trap

The trap is **`system = "x86_64-linux"` hardcoded** in a flake used on Apple silicon, or `builtins.currentSystem` inside a flake (impure). Enumerate systems. Use `builtins.currentSystem` only inside impure `nix-shell`, not in flakes.

The boring rule

- Enumerate systems with `lib.genAttrs`. 26.05 pin.
- Native remote builders for foreign Linux. Not laptop QEMU for releases.
- CI matrix for the OS humans actually use.
- Do not plan on `x86_64-darwin` past 26.05.
- Two archs = two cache objects = two `nix copy`s.

Try this

1. `nix flake show` and list systems.
2. On a Mac, `file $(nix develop --command which go)` — arm64 on Apple silicon.
3. `nix build .#packages.aarch64-linux.default --max-jobs 0` without a builder and read the error.
4. Note the 26.11 darwin x86_64 drop in the version appendix; do not add that system to new flakes.

Build Optimization and Performance Tuning

Build Optimization and Performance Tuning

Waiting on eval is not a personality trait. The boring default is: **substituters first, `auto-optimise-store`, filtered `src`, honest `max-jobs`, and a profiler only when `nix develop` takes minutes to *start*.**

Mental model

Slow	Fix
Download	Team cache, `builders-use-substitutes`
Compile	Remote builder; do not rebuild gcc locally
Eval	No IFD; `fileset`; one nixpkgs
Disk	GC window + `--optimise`

Laptops: default `max-jobs` is fine. An 8-core CI runner should say `max-jobs = 8`. A 64G builder with `max-jobs = 1` is a mis-copied gist.

`cores = 0` lets each job use all CPUs and oversubscribe. Prefer a number on a shared builder. Cache hits make `cores` irrelevant — check substituters before you tune jobs.

Worked examples

Case 1: NixOS 26.05 knobs

Save as `nix-build.nix`:

```
# nix-build.nix
{
  nix.settings = {
    max-jobs = 8;
    cores = 4;
    substituters = [
      "https://cache.nixos.org"
      "https://desk-cache.cachix.org"
    ];
    trusted-public-keys = [
      "cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY="
      "desk-cache.cachix.org-1:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
```

```
    ];
    auto-optimise-store = true;
    builders-use-substitutes = true;
  };
}
```

```
nix show-config | grep -E 'max-jobs|cores|substituters|auto-optimise'
```

You should see the team cache, not only `cache.nixos.org`. Missing `trusted-public-keys` is why substitutes are silently ignored.

Case 2: No IFD on the hot path

Bad — eval must **build** before it can parse:

```
# do not
{
  deskLib = import "${pkgs.fetchFromGitHub {
    owner = "desk";
    repo = "lib";
    rev = "abc123";
    hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
  }}/default.nix";
}
```

Fix — flake input or a vendored file:

```
# flake.nix
{
  inputs.desk-lib.url = "github:desk/lib/abc123";
  outputs = { self, nixpkgs, desk-lib }: {
    # import desk-lib as an input, not a fetch inside eval
  };
}
```

IFD is serial and hostile to flakes. `callPackage ./vendored.nix` is also fine.

Confirm you are not IFDing by accident:

```
nix flake show 2>&1 | grep -i 'building.*fetch' || echo 'no fetch during eval (good)'
```

A `building '/nix/store/…-source.drv'` line during `flake show` is IFD. Move that fetch to a flake input.

Case 3: Measure eval

```
time nix flake show
time nix eval .#packages.x86_64-linux.desk-api --raw
```

If `flake show` is minutes, you import too much at the top. Split packages; avoid reading the whole monorepo in `outputs`. Filter `src` with `lib.fileset` so a README does not re-eval Go.

Case 4: `--dry-run` after a docs edit

```
nix build .#desk-api --dry-run
echo x >> README.md
nix build .#desk-api --dry-run
```

Second list must not include gcc. If it does, `src` is unfiltered (`src = ./.;` at repo root).

Case 5: Heavy compiles belong on the builder

Chromium, Firefox, LLVM: `nix.buildMachines` + cache. Confirm with:

```
nix build nixpkgs#hello --max-jobs 0 --dry-run
```

`--max-jobs 0` means "do not build locally." If hello is not substituted, the cache is wrong — fix that before tuning `cores`. A laptop compiling LLVM is a missing substituter, not a missing `-j`.

```
nix build nixpkgs#hello --print-build-logs 2>&1 | grep -E 'copying path|building' | head
```

`copying path` from `cache.nixos.org` (or the team cache) is the win. `building` hello on a workstation means substituters are empty or unsigned.

```
free -h
nproc
```

Set `max-jobs` from RAM and cores you actually have during a **real** build, not from a blog.

The trap

The trap is **turning `max-jobs` down to 1 on a big box** because a blog mentioned RAM. Watch `free -h` during a real build; then set jobs. The other trap is eval-profiler cargo-cult without reading `--dry-run`. Filter `src` first.

The boring rule

- Cache, then jobs, then eval hygiene.
- No IFD on the hot path.
- `--dry-run` after trivia. Filtered `src`.
- Heavy derivations on the builder. `--max-jobs 0` on the laptop.
- `auto-optimise-store` on 26.05.

Try this

1. `nix show-config | grep max-jobs` and the substituter list.
2. Case 4 on a filtered package.
3. `time nix flake show` vs `time nix build nixpkgs#hello` when hello is cached.
4. Enable `auto-optimise-store`; `systemctl status nix-optimise.timer` if you also set the timer.

Debugging and Troubleshooting Production Issues

Debugging and Troubleshooting Production Issues

"Works on my laptop" is a missing `nix log`. The boring default is: **`nix log` → `journalctl` → `why-depends` → `--show-trace` → rollback — before editing a live unit**.

Mental model

Symptom	First tool
Build failed	`nix log /nix/store/…drv`
Eval failed	`--show-trace`, `nix repl .#`
Unit dead after switch	`systemctl status`, `journalctl -u`
Closure huge	`nix path-info -Shr`, `why-depends`
Will not boot	Previous generation in systemd-boot

`vim /etc/systemd/system/foo.service` is undone by the next switch (or by impermanence). Put the flag in Nix. Live drop-ins from `systemctl edit` are unmanaged state.

The log of a failed build lives in the store. CI can `nix log` the same drv hash the laptop saw. That is the whole point of a content-addressed failed build: the bytes of the log are the same everywhere.

Worked examples

Case 1: Build log

Save as `fail.nix`:

```
# fail.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.runCommand "desk-fail" { } ''
  echo boom >&2
  false
''
```

```
nix-build fail.nix
nix log $(nix-instantiate fail.nix)
```

```
nix build .#desk-api --print-build-logs
```

You should see `boom` in the log even after the build is gone from the terminal scrollback. On CI:

```
nix log /nix/store/…-desk-fail.drv
```

Case 2: Eval trace

```
nix eval .#nixosConfigurations.app-01.config.networking.hostName --show-trace
```

Read the **last** frame first. Infinite recursion is usually `rec` + `//` or `final.pkg.overrideAttrs` calling `prev` the wrong way.

```
error: infinite recursion encountered
       at /nix/store/…/overlays.nix:4:5
```

`nix repl .#` then `:lf .` and tab-complete the attr you meant. Do not add `builtins.trace` to production modules and leave it there.

Case 3: Unit after switch

```
systemctl status desk-api --no-pager
journalctl -u desk-api -n 80 --no-pager
systemctl cat desk-api
```

`systemctl cat` shows a store path:

```
# /nix/store/…-unit-desk-api.service/desk-api.service
[Service]
ExecStart=/nix/store/…-desk-api/bin/desk-api
```

A drop-in from `systemctl edit` appears as `/etc/systemd/system/desk-api.service.d/override.conf`. Delete it and encode the change in the flake. On an ephemeral root it is already gone after reboot — which is why "it worked until reboot" is this bug.

Case 4: Unexpected python in the OS closure

```
nix path-info -Shr /run/current-system | tail
nix why-depends /run/current-system nixpkgs#python3
```

Output (shape):

```
/nix/store/…-nixos-system-app-01-26.05
  /nix/store/…-desk-wrapper
      /nix/store/…-python3-3.12.x
```

Cut the edge (a wrapper, `propagatedBuildInputs`, a shebang). Re-switch. Measure `path-info -Shr` before and after. gcc in the **runtime** closure of `desk-api` is the same class of bug.

Case 5: Rollback then revert git

```
sudo nixos-rebuild list-generations | tail
sudo nixos-rebuild switch --rollback
git log -1 --oneline
git revert HEAD
colmena apply --dry-run --on app-01
```

Rollback without a git revert means the next Colmena apply re-breaks the box. The generation on metal and the flake in git must agree.

If the box will not boot, pick the previous generation in systemd-boot. Then SSH in and revert.

Diff two system closures when "what changed?" is the ticket:

```
nix-diff /run/current-system /nix/var/nix/profiles/system-43-link
```

Or, without `nix-diff`:

```
nix store diff-closures /run/booted-system /run/current-system
```

You should see unit files and store paths, not a wall of llvm.

If `switch` failed halfway:

```
systemctl --failed --no-pager
journalctl -b -p err --no-pager | tail
```

A failed unit after a successful eval is not a Nix bug — it is the unit. Fix the module, `switch` again. Do not `systemctl reset-failed` and walk away.

The trap

The trap is **`systemctl edit` + reboot "to see."** Unmanaged state. Impermanence eats it; a persistent root lies to the next apply. The other trap is screenshotting a red CI log instead of `nix log` on the drv.

The boring rule

- Store logs + journald. Not screenshots.
- `--show-trace` for eval. `why-depends` for size.
- Rollback first; bisect the flake second.
- No live drop-ins. `systemctl cat` must be a store path only.
- 26.05 `nix log` / `path-info` are enough to start.

Try this

1. Case 1: `runCommand` with `false`; `nix log` the failed drv.
2. `systemctl cat sshd` — find `/nix/store`.
3. `--rollback` after a hostname change on a lab VM, then revert git.
4. `why-depends hello` vs gcc — gcc must not be in the **runtime** closure.

Testing Strategies for Nix Configurations

Testing Strategies for Nix Configurations

`nixos-rebuild switch` on prod is not a test. The boring default is: **`nix flake check` on every PR, `pkgs.testers.runNixOSTest` for services that listen, and a VM that is not the laptop**.

Mental model

Level	Command	What it proves
Eval	`nix eval`, module types	The expression is a value
Build	`nix build .#desk-api`	The derivation realises
NixOS VM	`runNixOSTest`	A unit actually listens
Full gate	`nix flake check`	Every `checks.*` attr

A module that is `mkIf false` evals forever. A VM test that `curl`s the port catches the empty `wantedBy`.

VM tests need `kvm` (and usually `nixos-test`) on the builder. Darwin laptops offload them to a Linux remote builder. `checkPhase` inside a derivation cannot hit the network — mock or vendor fixtures.

Keep PR tests under a few minutes. Nightly runs the farm.

Worked examples

Case 1: Cheap flake checks

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk checks";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
```

```
    in
    {
      packages.${system}.desk-api = pkgs.hello;

      checks.${system}.fmt = pkgs.runCommand "desk-fmt" { } ''
        echo ok > $out
      '';

      checks.${system}.api = self.packages.${system}.desk-api;
    };
}
```

```
nix flake check -L
```

`-L` prints logs. Failures must be visible without SSH to the runner. Replace `hello` with the real `desk-api` package.

Case 2: NixOS VM test

Save as `test-desk.nix`:

```
# test-desk.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.testers.runNixOSTest {
  name = "desk-nginx";
  nodes.server = {
    services.nginx.enable = true;
    networking.firewall.allowedTCPPorts = [ 80 ];
    system.stateVersion = "26.05";
  };
  nodes.client = {
    system.stateVersion = "26.05";
  };
  testScript = ''
    start_all()
    server.wait_for_unit("nginx.service")
    client.succeed("curl -f http://server/")
  '';
}
```

```
nix-build test-desk.nix
```

Needs KVM. Wire the same attr into `checks.${system}.nginx` so `nix flake check` runs it in CI.

A failing assertion:

```
error: client failed: curl -f http://server/
```

Non-zero exit. That is the gate.

Case 3: CI job

Save as `.github/workflows/check.yml` (shape):

```yaml
# .github/workflows/check.yml
name: check
on: [pull_request]
jobs:
  flake:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: nix flake check -L
```

The runner must have Nix and, for Case 2, KVM (self-hosted Linux builder with `supportedFeatures = [ "kvm" "nixos-test" ]`). GitHub-hosted ubuntu without nested virt skips the VM check or offloads it.

Case 4: Hermetic `checkPhase`

```
# desk-api.nix fragment
{
  doCheck = true;
  checkPhase = ''
    # no curl to api.desk.internal
    go test ./...
  '';
}
```

If the test needs a fixture, vendor it in `src`. A check that needs prod credentials is not a check — it is a smoke probe you run from `ops-01` after deploy.

Case 5: What not to run on every PR

A 20-node mesh test on every docs typo is how people skip CI. Split:

When	What
Every PR	`fmt`, `.#desk-api`, one small `runNixOSTest`
Nightly	`linkFarm` of images, multi-node mesh
After Colmena	live `curl` from `ops-01`

The trap

The trap is **eval-only CI**. `nix flake check` that only type-checks modules never sees `wantedBy = [ ]`. Pair eval with at least one VM test per listening service, or you will discover the empty unit on prod.

The boring rule

- `flake check -L` on every PR. VM tests when a unit listens.
- KVM on the Linux builder. 26.05 `runNixOSTest`.
- No network in `checkPhase`.
- Small PR tests; big nightly.
- A test that needs prod credentials is a probe, not a check.

Try this

1. `nix flake check -L` on the desk flake.
2. Run Case 2 (needs KVM). Confirm `curl` succeeds.
3. Add `client.succeed("false")` and confirm non-zero exit.
4. Add a `checks` attr that is just `self.packages.${system}.desk-api` so a broken Go build fails the PR without a VM.

Maintenance, Upgrades, and Migration

Maintenance, Upgrades, and Migration

`nix flake update` on Friday without a plan is a weekend. The boring default is: **pin `nixos-26.05`, bump in a dedicated PR, `flake check` + one lab host, then Colmena — never bump `stateVersion` in that same PR**.

Mental model

Move	What changes
`nix flake update nixpkgs`	Same train, newer commit
`nixos-25.11` → `nixos-26.05`	Release upgrade; read the notes
`stateVersion`	Almost never

25.11 is EOL. New hosts: `"26.05"`. Old hosts: change the **input**, keep their birth `stateVersion`.

The 26.05 **channel** ships Nix **2.34**; this book's CLI baseline is **2.35+** via the installer or `nix.package`. Do not confuse those in the upgrade PR: bumping the channel does not magically bump the daemon.

Home Manager's `release-26.05` follows the same train. HM 24.11 + nixpkgs 26.05 is eval pain.

Worked examples

Case 1: Lock bump inside 26.05

Save as the existing `flake.nix` input (already on 26.05):

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
}
```

```
nix flake update nixpkgs
nix flake metadata
nix flake check -L
sudo nixos-rebuild dry-activate --flake .#desk-lab
```

Read which units restart. Switch the lab. Then Colmena `--on app-01`, then the rest. One commit, one PR, changelog the rev.

Case 2: Release upgrade

```
# flake.nix - was nixos-25.11
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  inputs.home-manager.url = "github:nix-community/home-manager/release-26.05";
  inputs.home-manager.inputs.nixpkgs.follows = "nixpkgs";
}
```

`git diff flake.lock` is huge. Run VM tests. Stage-1 systemd is **default** on 26.05 — if you still set `boot.initrd.systemd.enable` by hand, read that chapter before the fleet apply.

```
nix flake check -L
nixos-rebuild dry-activate --flake .#desk-lab
```

Case 3: `stateVersion` stays

Save as `hosts/app-01.nix` for a box born on 25.11:

```
# hosts/app-01.nix
{
  system.stateVersion = "25.11"; # born on 25.11; input is 26.05
}
```

Still correct. Do not “tidy” it to `"26.05"` in the lock-bump PR. Database dirs and module defaults then migrate on a surprise.

New metal:

```
# hosts/app-03.nix
{
  system.stateVersion = "26.05";
}
```

Birth, not channel.

Case 4: Home Manager follows the same train

```
# flake.nix
{
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
  inputs.home-manager.url = "github:nix-community/home-manager/release-26.05";
  inputs.home-manager.inputs.nixpkgs.follows = "nixpkgs";
}
```

```
nix flake metadata
```

One nixpkgs. `home.stateVersion` is also birth (`"25.11"` on an old user, `"26.05"` on a new one). Same rule as the system.

Case 5: Record the lock

Save as a changelog line, not a wiki:

```
# CHANGELOG.md
- 2026-09-10: nixpkgs 26.05.<rev> on desk-lab, then app-01 (Colmena).
  stateVersion unchanged. Nix CLI still 2.35+ via nix.package.
```

If you cannot name the lock in prod, you cannot roll back with confidence.

```
git show HEAD:flake.lock | jq -r '.nodes.nixpkgs.locked.rev'
```

Paste **that** rev, not "we updated nixpkgs." A Colmena rollback without this line in git is a generation the next apply will overwrite.

Do not bump Home Manager in the same PR as nixpkgs unless you must. Two diffs, two rollbacks:

```
nix flake update nixpkgs
nix flake check -L
# merge, then:
nix flake update home-manager
nix flake check -L
```

HM `release-26.05` already `follows` nixpkgs; updating HM alone is usually a module-fix, not a stdenv rebuild.

```
nix flake metadata --json | jq -r '.locks.nodes.nixpkgs.locked.rev'
```

Paste that rev into the changelog.

The trap

The trap is **changing `stateVersion` to match the channel**. Channel bump stateVersion bump. The other trap is `nix flake update` with no arguments on Friday, which also bumps every other input you forgot you had.

The boring rule

- 26.05 train. Lock PRs, not surprise `update`.
- Lab, then canary, then fleet.

- `stateVersion` / `home.stateVersion` is birth.
- HM release matches nixpkgs release. `follows`.
- Changelog the rev. Channel Nix 2.34 CLI 2.35+.

Try this

1. `nix flake metadata` — write the locked date and rev.
2. `dry-activate` after a lock bump; list units that would restart.
3. `git grep stateVersion` — none "updated" to 26.05 on old hosts without notes.
4. Skim 26.05 release notes for one module you use (Caddy, postgres, sshd).

Enterprise Operational Checklists

Enterprise Operational Checklists

A wiki of "remember to" is how generations rot. The boring default is: **scripts in the repo that run `flake check`, `restic`, and `colmena apply --dry-run` — not a PDF**.

Mental model

If it is not a command, it will not happen on Friday. Map each ritual to something you already have: GC, restic, sops rekey, Colmena, the 26.05 lock.

Commit `scripts/desk-daily.sh` and a systemd timer on `ops-01` if humans forget. The timer is a NixOS module, not a crontab someone added by hand.

```
ops-01 timer  →  scripts/desk-daily.sh
weekly timer  →  GC + restic snapshots
human         →  dry-run + canary before fleet
```

Worked examples

Case 1: Daily

Save as `scripts/desk-daily.sh`:

```
# scripts/desk-daily.sh
#!/usr/bin/env bash
set -euo pipefail
nix flake check -L
systemctl --failed --no-pager
df -h /nix /boot /persist
```

```
chmod +x scripts/desk-daily.sh
./scripts/desk-daily.sh
```

Non-zero `flake check` means do not deploy. Wire the same script as a CI job so ops and GitHub disagree less.

Save as `modules/desk-daily.nix` on `ops-01`:

```
# modules/desk-daily.nix
{
  systemd.services.desk-daily = {
    serviceConfig.Type = "oneshot";
    serviceConfig.ExecStart = "/persist/desk/scripts/desk-daily.sh";
  };
  systemd.timers.desk-daily = {
    wantedBy = [ "timers.target" ];
    timerConfig.OnCalendar = "09:00";
  };
}
```

Case 2: Weekly disk

Save as `scripts/desk-weekly.sh`:

```
# scripts/desk-weekly.sh
#!/usr/bin/env bash
set -euo pipefail
sudo nix-collect-garbage --delete-older-than 14d
sudo nix-store --optimise
systemctl status restic-backups-desk --no-pager || true
sudo restic snapshots | tail
```

Save as `modules/boot-limit.nix`:

```
# modules/boot-limit.nix
{
  boot.loader.systemd-boot.configurationLimit = 8;
  nix.gc.automatic = true;
  nix.gc.dates = "weekly";
  nix.gc.options = "--delete-older-than 14d";
}
```

Fourteen days of generations is enough to roll back a bad Friday. The ESP fills if you skip `configurationLimit`. The weekly script must `git -C /persist/desk pull --ff-only` or it checks last month's flake.

Case 3: Before fleet apply

```
git log -1 --oneline
nix flake metadata
nix flake check -L
colmena apply --dry-run --on app-01
colmena apply --on app-01
```

```
# watch journalctl -u desk-api on app-01
colmena apply --on @app
```

Never `@all` for a kernel bump. `db-01` is serial, after a restic snapshot.

Case 4: Advisory day

```
nix flake update nixpkgs
nix flake check -L
sudo nixos-rebuild dry-activate --flake .#desk-lab
# lab switch, then Case 3
```

Do not wait for the quarterly bump if nixpkgs already has the fix. Do not `nix flake update` with no arguments (that bumps every input). Changelog the rev.

Case 5: New host (checklist that is also git)

Save as `docs/new-host.md` in the same repo:

```
# docs/new-host.md
- Disko by-id + LUKS + @persist
- sops recipient + rekey
- stateVersion = "26.05"  (new metal only)
- WireGuard IP in the mesh module
- restic paths
- nixos-anywhere once
- Colmena targetHost + tag
```

Tick by merging a PR that adds the host — not by editing a spreadsheet. The PR is the checklist completing.

```
git grep -n 'stateVersion' -- '*.nix'
```

Every **new** host in that PR must be `"26.05"`. Old hosts keep their birth value. The checklist is wrong if the PR "tidies" `stateVersion` to match the channel.

The trap

The trap is **a 90-line Confluence page** and no `scripts/`. Five commands in git beat a beautiful table nobody runs. The other trap is a timer that runs `flake check` against a dirty `/persist` checkout nobody pulls.

The boring rule

- Commands in git. 26.05. sops. restic. Colmena.
- Dry-run, canary, then fleet. Never `@all` for kernels.
- 14-day GC. ESP `configurationLimit`.
- New hosts born on `"26.05"`.
- Advisories: lock bump the same day, changelog the rev.

Try this

1. Add Case 1; run it in CI as well as on ops.
2. Time `nix flake check`; if >10 minutes, split `checks`.
3. `restic snapshots` — newest < 48h.
4. Walk Case 5 on a throwaway VM and file the PR that adds it.

Fleet Deployments with Colmena

Fleet Deployments with Colmena

`for h in hosts; do nixos-rebuild --flake .#$h --target-host …; done` forgets a flag and leaves one box on last week's generation. The boring default is: **Colmena with one hive in the flake, SSH from `ops-01`, `--on` for canaries, and the same nixpkgs 26.05 lock the laptop uses**.

Mental model

Colmena evals a hive (host → modules + `deployment.targetHost`), realises each closure (locally or on a builder), copies it over SSH, and activates. Rollback is the previous NixOS generation on that host — not a restore CD.

```
ops-01  colmena apply --on app-01
                build + copy

        app-01  switch-to-configuration
```

Tags (`@app`, `@db`) are how you canary a role without typing every hostname. Parallel apply is for **stateless** boxes. Serial the node that owns the database.

`deployment.buildOnTarget = true` compiles on the host. The boring default is the opposite: build on `ops-01` (or the remote builder), copy the result. Laptops do not compile kernels.

Worked examples

Case 1: Hive in the flake

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk fleet hive";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    colmena = {
      meta = {
```

```
        nixpkgs = import nixpkgs { system = "x86_64-linux"; };
      };

      defaults = { pkgs, ... }: {
        system.stateVersion = "26.05";
        environment.systemPackages = [ pkgs.vim ];
        services.openssh.enable = true;
        services.openssh.settings.PasswordAuthentication = false;
        networking.firewall.enable = true;
      };

      app-01 = { name, ... }: {
        deployment.targetHost = "10.100.0.11";
        deployment.targetUser = "root";
        deployment.tags = [ "app" ];
        networking.hostName = name;
        imports = [ ./hosts/app-01.nix ];
      };

      app-02 = { name, ... }: {
        deployment.targetHost = "10.100.0.12";
        deployment.targetUser = "root";
        deployment.tags = [ "app" ];
        networking.hostName = name;
        imports = [ ./hosts/app-02.nix ];
      };

      db-01 = { name, ... }: {
        deployment.targetHost = "10.100.0.21";
        deployment.targetUser = "root";
        deployment.tags = [ "db" ];
        networking.hostName = name;
        imports = [ ./hosts/db-01.nix ];
      };
    };
  };
}
```

```
nix shell nixpkgs#colmena --command colmena eval -E '{ nodes }: builtins.attrNames nodes'
```

You should see `app-01`, `app-02`, `db-01`. Exact attr names follow the Colmena pin on 26.05 (`colmena --help`). Newer Colmena also accepts `colmenaHive` + `colmena.lib.makeHive` — if `colmena eval` says the flake attr moved, use that shape. The idea is still host → SSH target.

```
# hive extras
{
```

```
  deployment.allowLocalDeployment = true; # ops-01 applying itself
  deployment.keys."wg-priv" = {
    keyFile = "/run/secrets/wireguard/private";
    destDir = "/run/keys";
  };
}
```

Prefer sops-nix on the **target** over Colmena `deployment.keys` when you already have sops. Two secret planes is two rekeys.

Case 2: Canary, then the rest of the tag

```
colmena apply --on app-01
# watch desk-api metrics / journalctl -u desk-api
colmena apply --on @app
```

Do not `--on @all` for a kernel bump. `db-01` is a separate `--on db-01` after the apps are healthy.

Case 3: Dry-run before activate

```
colmena apply --dry-run --on app-01
```

Eval + build, no `switch`. If the dry-run fails, git is wrong — not the host.

```
[app-01] Built /nix/store/7q1a…-nixos-system-app-01-26.05
[app-01] (dry run; not activating)
```

Case 4: Build where the CPU is

Keep `deployment.buildOnTarget = false` (the default). Point the hive machine at the same remote builder as `nix.buildMachines` on `ops-01`. Confirm:

```
nix show-config | grep -E 'builders|max-jobs'
```

`--max-jobs 0` on the laptop forces remote realisation. A canary that compiles LLVM on `app-01` is a production incident.

Case 5: Activation failure is a generation, not a brick

```
[app-01] Built /nix/store/7q1a…-nixos-system-app-01-26.05
[app-01] Activation failed
```

The previous generation is still the boot default unless you passed a reboot-into-new-generation flag (`reboot` / `apply-local --reboot` on your pin). On the host:

```
ssh root@10.100.0.11 'systemctl is-system-running; nixos-rebuild list-generations | tail'
```

Roll back on the box (`nixos-rebuild switch --rollback` or Colmena's rollback flag on your pin), then revert the git commit so the hive and the metal agree.

The trap

The trap is **applying every host in parallel while a shared Postgres migration runs**. Stateless `desk-api` boxes can fan out. The stateful node is serial, after the migration is in the module and the backup snapshot exists.

The boring rule

- One hive, one 26.05 lock, SSH from `ops-01`.
- Canary `--on app-01`, then `--on @app`. Never `@all` for kernels.
- Dry-run first. Build off the target.
- Stateful hosts serial. Stateless hosts tagged.
- Rollback is the previous generation, then a git revert so the hive matches.
- One secret system (sops on the host). Colmena `deployment.keys` only if you do not have sops yet.

Try this

1. `nix shell nixpkgs#colmena --command colmena --help`.
2. Apply to a single lab VM with `--dry-run`, then without.
3. Break a unit on purpose, apply, roll back on the VM, revert git.
4. Write `docs/fleet.md`: which tags may run in parallel, which host is serial.

Lightweight MicroVMs with microvm.nix

Lightweight MicroVMs with microvm.nix

Full libvirt guests take minutes to iterate. The boring default for **local** multi-node tests is: **microvm.nix (or `runNixOSTest`) on nixpkgs 26.05; production still gets real machines or the capstone VMs**.

Mental model

`microvm.nix` is a NixOS module that boots a guest in hundreds of milliseconds, often sharing the host store via virtiofs. It is a **lab accelerator**, not a tenant isolator. Do not put untrusted workloads there without reading the project's trust model.

```
host flake
  nixosConfigurations.desk-micro
    microvm.hypervisor = "qemu"   # or cloud-hypervisor
    microvm.mem / vcpu
    volumes you listed - nothing else persists
```

`runNixOSTest` is the CI twin: headless, assertions, no SSH session. Keep both only when they answer different questions (human poke vs PR gate).

`/dev/kvm` must exist on the host. Nested virt on a cloud VM is often off — that is a missing feature, not a Nix bug.

Worked examples

Case 1: Guest in the flake

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk microVM lab";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    microvm.url = "github:astro/microvm.nix";
    microvm.inputs.nixpkgs.follows = "nixpkgs";
  };
```

```
  outputs = { self, nixpkgs, microvm }: {
    nixosConfigurations.desk-micro = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        microvm.nixosModules.microvm
        {
          microvm.hypervisor = "qemu";
          microvm.mem = 1024;
          microvm.vcpu = 2;
          networking.hostName = "desk-micro";
          services.openssh.enable = true;
          services.openssh.settings.PasswordAuthentication = false;
          users.users.root.openssh.authorizedKeys.keys = [
            "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAI… deskadmin@ops-01"
          ];
          system.stateVersion = "26.05";
        }
      ];
    };
  };
}
```

`follows` keeps the guest on the same 26.05 as the host flake.

Case 2: Run the declared runner

```
nix run .#nixosConfigurations.desk-micro.config.microvm.declaredRunner
```

(The exact run attr is in microvm.nix docs for your pin.) You should get a console. Log in, then:

```
hostname
systemctl is-system-running
systemctl status sshd --no-pager
halt
```

If QEMU exits immediately, check `ls -l /dev/kvm` and that your user is in `kvm`.

```
ls -l /dev/kvm
id -nG | tr ' ' '\n' | grep -x kvm || echo 'not in kvm group - log out after extraGroups'
```

Nested virt on a cloud workstation: the hypervisor UI must enable it. `ls /dev/kvm` missing is not a Nix eval error.

Case 3: Persist nothing unless listed

A default microVM is as ephemeral as a tmpfs root. If a test needs `/persist`, declare a volume in the module:

```
# volumes.nix
{
  microvm.volumes = [
    {
      image = "persist.img";
      mountPoint = "/persist";
      size = 1024;
    }
  ];
}
```

Otherwise the next `nix run` starts clean — which is what you want for a lab.

Case 4: Host virtualisation (workstation)

Save as `hosts/ops-lab.nix` (fragment of the workstation):

```
# hosts/ops-lab.nix
{
  virtualisation.libvirtd.enable = true;
  users.users.deskadmin.extraGroups = [ "libvirtd" "kvm" ];
}
```

```
ls -l /dev/kvm
groups
```

`kvm` in the group list, `crw-rw----` on `/dev/kvm`. Nested virt on a cloud workstation: enable it in the hypervisor UI or pick a metal builder.

Case 5: microVM vs `runNixOSTest`

Need	Tool
SSH in, poke Caddy, read journals	microVM on the laptop
PR must fail if nginx does not listen	`pkgs.testers.runNixOSTest` in `checks`
Production desk-api	Colmena to real hosts

Do not maintain both for the same scenario without a reason. CI does not SSH into your laptop guest.

The trap

The trap is **running production `desk-api` in a microVM on someone's laptop**. Prod is Colmena to real hosts (or the capstone VMs). microVMs die when the lid closes.

The boring rule

- 26.05 + `microvm.inputs.nixpkgs.follows`.
- QEMU/KVM for labs. `runNixOSTest` for CI.
- No production tenants on a laptop hypervisor.
- `stateVersion = "26.05"` on new guests.
- Share the store only when you understand the trust model. Volumes you did not list do not exist.

Try this

1. Enable KVM, run Case 1, `hostname` inside the guest.
2. `systemctl status sshd` inside it, then `halt`.
3. Compare wall-clock boot to a libvirt NixOS guest on the same box.
4. Add a `checks.x86_64-linux runNixOSTest` that covers the same `sshd` enablement so CI does not need the laptop.

Content-Addressed Nix Internals

Content-Addressed Nix Internals

Everyday Nix is **input-addressed**: the `.drv` hash names the output. **Content-addressed** (CA) derivations hash the **bytes**. The boring default on 26.05 is: **input-addressed builds, FODs (`fetchurl`, `vendorHash`) for the CA you already use, `nix-store --optimise` for disk — not experimental CA on the laptop**.

Mental model

Kind	Output path depends on	Daily
Input-addressed	The `.drv` (builder, env, input paths)	Packages, NixOS
FOD	Declared `outputHash`	Fetchers, cargo/go vendor
Experimental CA	Hash of the built bytes	Hydra labs

A comment in a `runCommand` changes the `.drv` even if `cat $out` is identical. That wastes cache. It also makes "why did this rebuild?" answerable. Live with it.

Nix 2.35 lazy flake copy is **eval** performance. It is not CA builds. Do not mix them in an incident doc. The 26.05 channel's Nix 2.34 vs this book's CLI 2.35+ does not change the addressing model you ship.

Worked examples

Case 1: FOD — identity is the hash

Save as `fod.nix`:

```
# fod.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.fetchurl {
  url = "https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz";
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build fod.nix
```

Mismatch prints `got:`. Paste that into `hash`. Same `hash`, different URL (mirror) → same store path. That is the CA you already depend on: `vendorHash`, `cargoHash`, `fetchurl`.

`outputHashMode` is `"flat"` (one file, `fetchurl`) or `"recursive"` (a directory NAR, `fetchFromGitHub`, `vendorHash`). Mixing them is a hash mismatch that looks like a wrong URL. If `got:` is `sha256-` of a NAR and you expected a single-file hash, you used the wrong mode — not a bad mirror.

Case 2: Comment rebuilds an input-addressed path

Save as `banner.nix`:

```
# banner.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.runCommand "desk-banner" { } ''
  echo desk > $out
  # lab note
''
```

```
nix-instantiate banner.nix
# change the comment, instantiate again
nix-instantiate banner.nix
```

Two `.drv` hashes. `diff` the two `$out` files after `nix-build` — same bytes, different names. Normal for input-addressed builds. Do not "fix" this with `__contentAddressed`.

Case 3: `nix show-derivation`

```
nix show-derivation $(nix-instantiate -E '(import <nixpkgs> {}).hello') | head
nix show-derivation $(nix-build fod.nix --no-out-link) | jq '.[].outputs'
```

FODs have `outputHash` / `outputHashAlgo`. `hello` does not. If you are staring at a path and do not know which family it is, this is the check.

```
"outputs": {
  "out": {
    "hash": "…",
    "hashAlgo": "r:sha256"
  }
}
```

Case 4: Disk without CA — optimise

```
sudo nix-store --optimise
df -h /nix
```

Hard-link identical **files** inside distinct input-addressed paths. No experimental flags.

Save as `nix-optimise.nix`:

```
# nix-optimise.nix
{
  nix.settings.auto-optimise-store = true;
  nix.optimise.automatic = true;
}
```

14-day GC plus this is the disk plan. Experimental CA is not.

Case 5: Do not enable CA on the desk

```
# do not
{
  nix.settings.experimental-features = [ "ca-derivations" ];
}
```

unless the whole team **and** the cache agree. nixpkgs 26.05 still assumes input-addressed for almost everything you ship. Substituters miss, support assumes input-addressed, and a one-line stdenv change becomes a debugging career.

A recursive FOD (directory) looks like this — still not experimental CA:

```
# fod-git.nix
{ pkgs ? import <nixpkgs> { } }:

pkgs.fetchFromGitHub {
  owner = "NixOS";
  repo = "nixpkgs";
  rev = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
  hash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
}
```

```
nix-build fod-git.nix
nix show-derivation $(nix-build fod-git.nix --no-out-link) | jq '.[].outputs.out.hashAlgo'
```

You want `r:sha256` (recursive). `fetchurl` of a `.tar.gz` is usually flat. Same `got:` paste workflow.

The trap

The trap is **`__contentAddressed = true` on a production module** "to save disk." Use GC + `--optimise`. The other trap is calling Nix 2.35 lazy trees "content-addressed Nix" in a postmortem — different layer.

The boring rule

- FODs for fetches. Input-addressed for builds.
- `--optimise` + 14-day GC for disk.
- `show-derivation` when a path's kind is unclear.
- 2.35 lazy copy CA.
- No `ca-derivations` on workstation flakes.

Try this

1. Case 3: compare `hello` vs a `fetchurl` for `outputHash`.
2. Case 2: two instantiations; `diff` the `.drv` files.
3. `git grep contentAddressed` / `ca-derivations` — empty on the desk.
4. `nix-store --optimise` on a lab; note `df` delta.

Production Capstone Lab

Production Architecture and Threat Model

Production Architecture and Threat Model

Toy VMs taught the pieces. The capstone is one **desk fleet** on NixOS 26.05: five roles, WireGuard, ephemeral workers, sops-nix, Colmena. The boring default is: **write the threat model first, then the flake — not a mesh of tools looking for a problem**.

Mental model

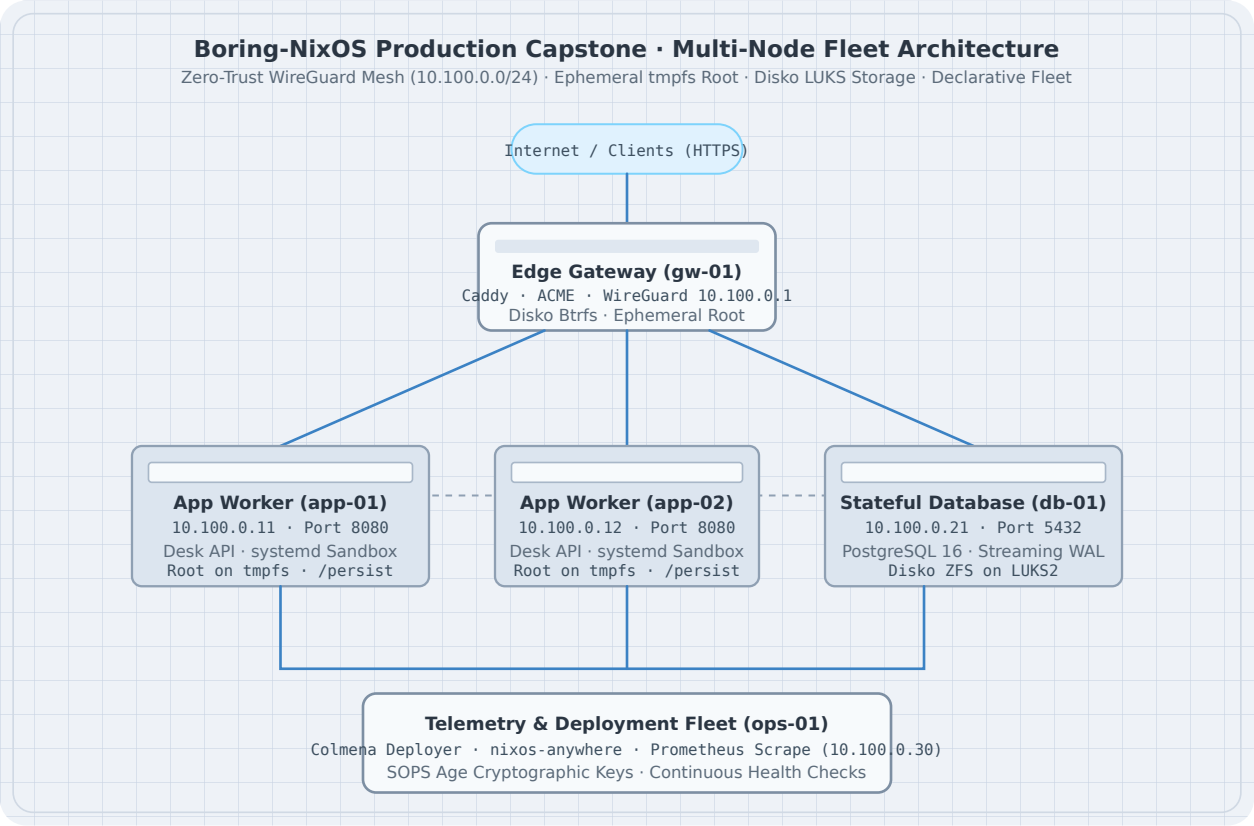

Figure 1: Production Capstone Fleet Architecture

Host	Role	Persist
`gw-01`	Caddy, ACME, nftables; WireGuard `10.100.0.1`	certs, Caddy state
`app-01`, `app-02`	desk-api; tmpfs root; `DynamicUser`	nothing (ephemeral)

Host	Role	Persist
db-01	PostgreSQL; Disko LUKS + Btrfs	/persist
ops-01	Colmena, Prometheus, sops key broker	hive checkout, keys

Invariants:

1. No secret bytes in git or /nix/store.
2. Store is read-only. Payloads in /etc die on reboot (workers).
3. WireGuard for all east-west. No DB port on the public NIC.
4. flake.lock on **nixos-26.05**; rollback is a generation, not a restore CD.
5. Backups of /persist (restic), not of /nix/store.

Public surface: TCP 443 on gw-01 (and UDP 51820 on every peer). SSH from the office VPN or the ops-01 jump only.

Worked examples

Case 1: Inventory in the flake

Save as flake.nix:

```
# flake.nix
{
  description = "Desk production fleet";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }:
    let
      mk = name: modules: nixpkgs.lib.nixosSystem {
        system = "x86_64-linux";
        modules = [
          { networking.hostName = name; system.stateVersion = "26.05"; }
        ] ++ modules;
      };
    in
    {
      nixosConfigurations = {
        gw-01  = mk "gw-01"  [ ./hosts/gw-01.nix ];
        app-01 = mk "app-01" [ ./hosts/app-01.nix ];
        app-02 = mk "app-02" [ ./hosts/app-02.nix ];
        db-01  = mk "db-01"  [ ./hosts/db-01.nix ];
        ops-01 = mk "ops-01" [ ./hosts/ops-01.nix ];
```

```
      };
    };
}
```

Five names. Five `system.stateVersion = "26.05"` because they are born here. Do not copy a `stateVersion` from a laptop that was installed on 24.11.

```
nix flake show
```

You should see five `nixosConfigurations`.

Save as `hosts/gw-01.nix` (shape — later chapters fill Caddy and WireGuard):

```
# hosts/gw-01.nix
{
  imports = [ ../modules/base-system.nix ../modules/wireguard-mesh.nix ];
  networking.firewall.allowedTCPPorts = [ 80 443 ];
  networking.firewall.allowedUDPPorts = [ 51820 ];
  # services.caddy … in the hardened-network chapter
}
```

If a port is not in `docs/threats.md`, it is not in this module.

Case 2: Trust boundary in git

Save as `docs/threats.md`:

```
# docs/threats.md
Public NIC: 443/tcp on gw-01, 51820/udp on every peer.
SSH: office VPN or ops-01 jump. No 22/tcp from 0.0.0.0/0.
Postgres: 5432/tcp on 10.100.0.21, wg0 only.
desk-api: 8080 on wg0; Caddy reverse-proxies from gw-01.
Attacker with a stolen app-01 disk: tmpfs, no secrets.
Attacker with a stolen db-01 disk: LUKS; restic passphrase is not on the disk.
```

If a port is not in this file, the firewall and the cloud SG must not open it.

```
git grep -n 'allowedTCPPorts\|allowedUDPPorts' -- '*.nix'
```

Every port in that grep lands in `docs/threats.md` or it is a bug. `0.0.0.0/0` on 22 is a bug even if the module lists 22.

Case 3: Secret inventory

Secret	Mechanism	Lives
WireGuard private keys	sops-nix → `/run/secrets`	encrypted in git

Secret	Mechanism	Lives
Postgres password	sops-nix `EnvironmentFile`	encrypted in git
ACME account / certs	Caddy module + `/persist`	`gw-01` persist
Colmena SSH key	`ops-01` persist	**not** the flake
restic password	sops-nix on `db-01` / `ops-01`	encrypted in git

A secret that is only in a password manager is a secret that will not survive the next hire.

Case 4: Failure drill list

Failure	Response
Lose `app-01`	traffic already on `app-02`; nixos-anywhere a replacement; Colmena
Lose `db-01`	restic restore onto a new Disko disk; Colmena; check `pg_isready` on `wg0`
Bad generation on `app-01`	previous generation / Colmena rollback; revert git
Lose `gw-01`	ACME + persist restore; DNS TTL is why you kept it short
Stolen laptop with ops checkout	sops age key is **not** on the laptop; rotate the Colmena SSH key

Write the restic repository URL in `docs/threats.md`. Not the password.

Case 5: What this capstone is not

Not multi-region Kubernetes. Not Vault HA. Not a service mesh. Those are later, optional, and they each add a new trust boundary. If the five hosts rebuild from git + restic, you passed.

Order of the remaining capstone chapters: Disko + impermanence, WireGuard, application stack, Colmena + restic. Do not start with Istio.

The trap

The trap is **starting with three clouds and a mesh** before SSH keys persist across reboot. Finish Disko, impermanence, sops, WireGuard, Colmena, restic. In that order. A threat model that lists “Zero Trust” and does not list `/persist` is a slide, not an ops doc.

The boring rule

- 26.05 lock. Five hosts. One mesh. One secret system (sops).
- Public surface is 443 (+ 22 only from a tight SG / VPN).
- `/persist` is the backup. Store is the cache.
- Rollback is a generation.
- Write the threat model in git next to the flake.

Try this

1. Copy the host table into `docs/threats.md`. Fill module file names.
2. List every port you think is open; compare to firewall + cloud SG in later chapters.
3. Name the restic repository in `docs/threats.md`.
4. Confirm `stateVersion` will be `"26.05"` on new metal — not copied from an old laptop.

Disko Storage and Ephemeral Base System

Disko Storage and Ephemeral Base System

Every node in our production fleet must deploy onto clean physical or cloud storage without manual partitioning tools. In this chapter, we create the unified **Disko** storage specification and configure the **ephemeral root filesystem with Impermanence**.

Mental model

1. **Unified disk module (`disk-layout.nix`):** Disko formats the physical NVMe or virtual disk into an EFI system partition (`/boot`) and a LUKS2 encrypted partition. Metal uses `/dev/disk/by-id/…`; `lib.mkDefault "/dev/nvme0n1"` is a lab fallback, not a fleet device.
2. **Btrfs subvolumes on LUKS:** Inside the decrypted container (`/dev/mapper/<luks.name>`), Btrfs subvolumes are created for `/nix` (the store), `/persist` (stateful data), and `/swap`. The LUKS **name** is the Stage-1 mapper name. Disko `name = "crypted"` must match `boot.initrd.luks.devices.crypted` (Disko usually emits this). A mismatch and systemd Stage 1 waits on `/sysroot` forever.
3. **RAM root (`tmpfs`):** The root filesystem (`/`) is mounted in RAM (`mode=755`, size you can afford). Unmounted state is discarded on power-down.
4. **Impermanence persistence:** System state required across reboots (SSH host keys, machine-id, WireGuard identity, sops keys) is bound into `/persist`. `/persist` is `neededForBoot = true` so Stage 1 mounts it before activation tries to link `machine-id`.

Worked examples

Case 1: The Production Disko Module

Save as `modules/disk-layout.nix`:

```
# modules/disk-layout.nix
{ lib, ... }:

{
  disko.devices = {
    disk.main = {
      type = "disk";
      # Lab fallback. Per-host modules override with /dev/disk/by-id/nvme-….
      device = lib.mkDefault "/dev/nvme0n1";
      content = {
        type = "gpt";
```

```
      partitions = {
        ESP = {
          priority = 1;
          name = "ESP";
          start = "1M";
          end = "1024M";
          type = "EF00";
          content = {
            type = "filesystem";
            format = "vfat";
            mountpoint = "/boot";
            mountOptions = [ "umask=0077" ];
          };
        };
        luks = {
          size = "100%";
          content = {
            type = "luks";
            name = "crypted";
            settings.allowDiscards = true;
            content = {
              type = "btrfs";
              extraArgs = [ "-f" ];
              subvolumes = {
                "/nix" = {
                  mountpoint = "/nix";
                  mountOptions = [ "compress=zstd" "noatime" ];
                };
                "/persist" = {
                  mountpoint = "/persist";
                  mountOptions = [ "compress=zstd" "noatime" ];
                };
                "/swap" = {
                  mountpoint = "/swap";
                  swap.swapfile.size = "8G";
                };
              };
            };
          };
        };
      };
    };
  };

  nodev = {
    "/" = {
```

```
        fsType = "tmpfs";
        mountOptions = [ "defaults" "size=4G" "mode=755" ];
      };
    };
  };

  # Disko emits fileSystems."/persist" from the subvolume mountpoint.
  # Stage 1 must mount it before activation links machine-id / SSH keys.
  fileSystems."/persist".neededForBoot = true;
}
```

`ls -l /dev/disk/by-id/` on the box (or rescue). Paste `nvme-eui.*` or `ata-*`, not a `-partN` suffix and not a `wwn-` alias you did not verify. Override per host:

```
# hosts/app-01/disk.nix
{
  imports = [ ../../modules/disk-layout.nix ];
  disko.devices.disk.main.device =
    "/dev/disk/by-id/nvme-VENDOR_MODEL_SERIAL";
}
```

LUKS `name = "crypted"` is `/dev/mapper/crypted` in Stage 1. Do not rename it in one file and leave `boot.initrd.luks.devices` on the old name.

Case 2: The Impermanence Persistent State Contract

Save as `modules/base-system.nix`:

```
# modules/base-system.nix
{ inputs, pkgs, ... }:

{
  imports = [
    inputs.impermanence.nixosModules.impermanence
    ./disk-layout.nix
  ];

  boot.loader.systemd-boot.enable = true;
  boot.loader.efi.canTouchEfiVariables = true;
  boot.loader.systemd-boot.configurationLimit = 8;

  fileSystems."/persist".neededForBoot = true;

  # Persistent state mapping - if it is not listed, it dies at reboot.
  environment.persistence."/persist" = {
    hideMounts = true;
```

```
    directories = [
      "/var/log"
      "/var/lib/nixos"
      "/var/lib/systemd/coredump"
      "/var/lib/systemd/timers"
      "/var/lib/sops-nix"
    ];
    files = [
      "/etc/machine-id"
      "/etc/ssh/ssh_host_ed25519_key"
      "/etc/ssh/ssh_host_ed25519_key.pub"
    ];
  };

  # SSH configuration
  services.openssh = {
    enable = true;
    settings = {
      PasswordAuthentication = false;
      KbdInteractiveAuthentication = false;
      PermitRootLogin = "prohibit-password";
    };
  };

  system.stateVersion = "26.05";
}
```

Validating the base configuration:

```
nix eval .#nixosConfigurations.app-01.config.disko.devices.nodev."/".fsType
```

```
"tmpfs"
```

A new SSH host key every boot means sops cannot decrypt. Persist the ed25519 host key **and** `/var/lib/sops-nix`. WireGuard private keys belong in sops at `/run/secrets`, not as leftover files on tmpfs.

Case 3: `neededForBoot` is a Stage-1 mount, not a comment

```
nix eval .#nixosConfigurations.app-01.config.fileSystems.\"/persist\".neededForBoot
nix eval .#nixosConfigurations.app-01.config.boot.initrd.luks.devices.crypted.device
```

```
true
"/dev/disk/by-partlabel/disk-main-luks"
```

(Exact LUKS device path depends on Disko's GPT labels.) If `neededForBoot` is `false` or missing, activation looks for `/persist/etc/machine-id` before the subvolume is mounted. Emergency shell. Set it in the **same** module that owns Disko so a host cannot import the layout and forget the flag.

Case 4: Wrong disk, wrong mapper

```
ls -l /dev/disk/by-id/ | rg 'nvme|ata' | rg -v part
lsblk -o NAME,SIZE,MODEL,SERIAL,MOUNTPOINT
```

`/dev/nvme0n1` on rescue is often the installer USB. `by-id` includes the serial. Two hosts sharing `disk-layout.nix` without a per-host `device` override will format whichever disk is `nvme0n1` on that firmware.

Rename LUKS in Disko (`name = "cryptroot"`) only if **every** reference follows: `fileSystems."/nix".device`, Stage-1 `systemd-cryptsetup@cryptroot`, and any `crypttab` extra. The Stage-1 chapter uses `/sysroot` inside the initrd — a oneshot that writes `/persist` in Stage 1 is writing the initrd overlay unless `/persist` is already mounted under `/sysroot/persist`.

Case 5: Format is once; mounts are every boot

```
# lab VM only - destroys the disk
nix run github:nix-community/disko -- --mode destroy,format,mount ./modules/disk-layout.nix
mount | rg 'tmpfs on / | /nix | /persist | /boot'
```

`--mode mount` is the non-destructive path (installer / recovery). `--mode destroy,format,mount` is `nixos-anywhere`'s Disko phase. The running system never re-runs format: NixOS just mounts what Disko already described in `fileSystems.*`.

The trap

The trap is **failing to mark `/persist` as `neededForBoot = true`**. During early boot, the initramfs must mount `/persist` before systemd starts generating symlinks for machine-id and SSH keys. Always ensure `/persist` is mounted before root pivot.

The other trap is **one `device = "/dev/nvme0n1"` for the fleet**. Disk names move. Per-host `by-id`. The third is a LUKS mapper rename that Stage 1 still calls `crypted`.

The boring rule

- Model all partition tables and filesystems in Disko. Same file for `nixos-anywhere` and for `fileSystems`.
- `by-id` on metal. `mkDefault` + per-host override. Never a shared `nvme0n1`.
- LUKS `name` is the Stage-1 mapper. Do not drift.
- Keep `/` on tmpfs. `neededForBoot = true` on `/persist`.

- Explicitly list persistent files and directories (host keys, sops, logs, nixos uid map).

Try this

1. `nix eval fileSystems."/persist".neededForBoot` on `app-01`. Must be `true`.
2. `nix eval` the LUKS mapper name and confirm it is `crypted` (or the name you chose everywhere).
3. `ls -l /dev/disk/by-id` on a lab box; write the id you would put in `hosts/app-01/disk.nix`. Do not format that disk.
4. Run `nix run github:nix-community/disko -- --mode mount ./modules/disk-layout.nix` only in a test VM. `mount | grep tmpfs`.

Hardened Networking and WireGuard Mesh

Hardened Networking and WireGuard Mesh

Public IPs should not see Postgres. The boring default is: **WireGuard mesh `10.100.0.0/24`, keys via sops on `/run/secrets`, nftables so `5432` exists only on `wg0`.**

Mental model

Node	wg0
gw-01	10.100.0.1
app-01 / app-02	10.100.0.11 / .12
db-01	10.100.0.21
ops-01	10.100.0.30

Public: UDP **51820** on every peer; TCP **80/443** only on `gw-01`. Everything else: `wg0` or drop.

Private keys never enter `/nix/store`. `privateKeyFile` points at sops.

`persistentKeepalive = 25` keeps NAT mappings alive toward cloud UDP.

Worked examples

Case 1: Shared mesh module

Save as `modules/wireguard-mesh.nix`:

```
# modules/wireguard-mesh.nix
{ config, lib, ... }:

{
  networking.firewall.allowedUDPPorts = [ 51820 ];

  networking.wireguard.interfaces.wg0 = {
    ips = [ config.cluster.nodeIp ];
    listenPort = 51820;
    privateKeyFile = config.sops.secrets."wireguard/private".path;
    peers = [
      {
        publicKey = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
        allowedIPs = [ "10.100.0.1/32" ];
```

```
        endpoint = "203.0.113.10:51820";
        persistentKeepalive = 25;
      }
      {
        publicKey = "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB=";
        allowedIPs = [ "10.100.0.11/32" ];
        endpoint = "203.0.113.11:51820";
        persistentKeepalive = 25;
      }
      {
        publicKey = "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC=";
        allowedIPs = [ "10.100.0.21/32" ];
        endpoint = "203.0.113.21:51820";
        persistentKeepalive = 25;
      }
    ];
  };
}
```

Replace `AAA…` with real **public** keys. Private keys stay in sops.

Each peer must include **itself** in `ips` and every **other** node in `peers` with `/32` `allowedIPs`. A missing self IP is a silent blackhole. `endpoint` is the **public** UDP address; omit `endpoint` only for peers that only receive (roaming laptop) and then they need `persistentKeepalive` toward a fixed peer.

```
# cluster option (tiny)
{ lib, ... }:
{
  options.cluster.nodeIp = lib.mkOption {
    type = lib.types.str;
    description = "This host's wg0 CIDR, e.g. 10.100.0.11/24";
  };
}
```

Case 2: Database host — Postgres only on `wg0`

Save as `hosts/db-01.nix`:

```
# hosts/db-01.nix
{
  imports = [
    ../modules/base-system.nix
    ../modules/wireguard-mesh.nix
  ];
```

```
  cluster.nodeIp = "10.100.0.21/24";

  networking.firewall = {
    enable = true;
    interfaces.wg0.allowedTCPPorts = [ 5432 ];
    interfaces.eth0.allowedTCPPorts = [ ];
  };
}
```

```
nix eval .#nixosConfigurations.db-01.config.networking.firewall.interfaces.wg0.allowedTCPPor
```

Output:

```
[ 5432 ]
```

Case 3: Gateway — HTTP public, API only over mesh

```
# hosts/gw-01.nix fragment
{
  cluster.nodeIp = "10.100.0.1/24";
  networking.firewall.allowedTCPPorts = [ 80 443 ];
  networking.firewall.interfaces.wg0.allowedTCPPorts = [ ];
}
```

Caddy binds `0.0.0.0:443`. It **proxies to** `10.100.0.11:8080`, which is not open on the public NIC of `app-01`.

Case 4: App node does not open 8080 publicly

```
{
  cluster.nodeIp = "10.100.0.11/24";
  networking.firewall.interfaces.wg0.allowedTCPPorts = [ 8080 ];
}
```

Health checks from Caddy use `10.100.0.11`. A scanner on the cloud IP should not see 8080.

Case 5: Prove the path

On `ops-01` (once deployed):

```
sudo wg show
ping -c 1 10.100.0.21
nc -zv 10.100.0.21 5432
```

From a laptop **not** on the mesh, `nc` to the public IP port 5432 should fail. That is the test, not a green `wg show` alone.

```
sudo wg show wg0 dump
ip route get 10.100.0.21
```

Handshake age of minutes is NAT; `persistentKeepalive = 25`. Handshake never: wrong pubkey, UDP 51820 blocked, or `allowedIPs` does not contain that `/32`.

The trap

The trap is **`privateKey = "…" in Nix`**. World-readable store. `privateKeyFile` + sops. Public keys in git are fine.

The other trap is `allowedIPs = [ "0.0.0.0/0" ]` turning the mesh into a default route. Use `/32` per peer unless you **mean** to route the internet through `gw-01`.

The boring rule

- Mesh IPs in git. Private keys in sops → `/run/secrets`.
- `5432` / `8080` on `wg0` only.
- Public: 80/443 on gateway, UDP 51820 on peers.
- `/32` `allowedIPs` unless you are building a router.
- `persistentKeepalive = 25` behind NAT.

Try this

1. Eval Case 2; confirm `eth0` TCP list is empty.
2. Generate a keypair with `wg genkey | tee priv | wg pubkey`; put **only** the pubkey in the module.
3. Two VMs, one peer each; `ping` over `wg0`.
4. `ss -lntp` on `db-01` — Postgres on `10.100.0.21:5432`, not `0.0.0.0`.

Distributed Application Stack and Secrets

Distributed Application Stack and Secrets

Storage and the mesh exist. Now the workload: **Caddy + ACME on `gw-01`, sandboxed `desk-api` on the apps, PostgreSQL on `db-01`, sops-nix for the password**. The boring default is: **TLS at the edge, `EnvironmentFile` for secrets, `DynamicUser` + `ProtectSystem=strict` on the API**.

Mental model

```
Internet --443--> Caddy (gw-01)
                      reverse_proxy 10.100.0.11:8080, .12:8080

                desk-api  --wg0-->  Postgres 10.100.0.21

                      /run/secrets/db_password  (sops, tmpfs)
```

Caddy talks HTTP to the apps **on the mesh**. Apps do not terminate TLS. Postgres does not listen on the public NIC (previous chapter).

`MemoryDenyWriteExecute` is fine for `CGO_ENABLED=0` Go. Drop it if you add a JIT.

Worked examples

Case 1: Caddy on the gateway

Save as `hosts/gw-01.nix`:

```
# hosts/gw-01.nix
{
  services.caddy = {
    enable = true;
    email = "ops@desk.internal";
    # Lab: Let's Encrypt staging so you do not burn production rate limits.
    # Production: acmeCA = null; (26.05 default) so Caddy can fall back issuers.
    acmeCA = "https://acme-staging-v02.api.letsencrypt.org/directory";
    virtualHosts."api.desk.internal".extraConfig = ''
      reverse_proxy 10.100.0.11:8080 10.100.0.12:8080 {
        lb_policy round_robin
        health_uri /healthz
```

```
        health_interval 5s
        health_timeout 2s
      }
    '';
  };

  networking.firewall.allowedTCPPorts = [ 80 443 ];

  # Ephemeral root: ACME private keys live here. Miss this and every reboot
  # is a new account + a new cert (and a rate-limit event).
  environment.persistence."/persist".directories = [ "/var/lib/caddy" ];
}
```

ACME HTTP-01 needs 80/443 reachable on `gw-01`. DNS `api.desk.internal` → that public IP. `reverse_proxy` to **mesh** IPs (`10.100.0.11/12`), not `127.0.0.1` (the API is not on the gateway).

Caddy's ACME is **not** `security.acme`. Do not enable both for the same name unless you set `virtualHosts.<name>.useACMEHost` and point Caddy at lego's files. The boring path is Caddy's built-in ACME + persist `dataDir` (`/var/lib/caddy`). DNS-01 (Cloudflare etc.) belongs in `globalConfig` with `{$API_TOKEN}` and `services.caddy.environmentFile` from sops — never the token in the Caddyfile store path.

After the lab cert looks right in `curl -vI https://api.desk.internal`, set `acmeCA = null;` and switch. A staging cert in a browser is "not trusted"; that is success for the lab.

Case 2: Sandboxed API

Save as `modules/desk-service.nix`:

```
# modules/desk-service.nix
{ config, pkgs, lib, ... }:

let
  appPkg = pkgs.buildGoModule {
    pname = "desk-api";
    version = "2.0.0";
    src = lib.cleanSource ./src;
    vendorHash = "sha256-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=";
    env.CGO_ENABLED = "0";
  };
in
{
  sops.secrets.db_password = { };

  systemd.services.desk-api = {
    description = "Desk API";
    after = [ "network-online.target" ];
```

```
    wants = [ "network-online.target" ];
    wantedBy = [ "multi-user.target" ];
    environment = {
      PORT = "8080";
      DB_HOST = "10.100.0.21";
      DB_USER = "desk_user";
      DB_NAME = "desk_production";
    };
    serviceConfig = {
      ExecStart = "${appPkg}/bin/desk-api";
      DynamicUser = true;
      # dotenv (DB_PASSWORD=…). Alternative: LoadCredential=db:/run/secrets/…
      # and read CREDENTIALS_DIRECTORY/db in the app - never argv.
      EnvironmentFile = config.sops.secrets.db_password.path;
      ProtectSystem = "strict";
      ProtectHome = true;
      PrivateTmp = true;
      PrivateDevices = true;
      NoNewPrivileges = true;
      RestrictRealtime = true;
      RestrictAddressFamilies = [ "AF_INET" "AF_INET6" "AF_UNIX" ];
      MemoryDenyWriteExecute = true;
    };
  };
}
```

`db_password` file is dotenv (`DB_PASSWORD=…`), not a CLI flag. For `DynamicUser`, leave the secret **root-owned** `0400` and readable via `EnvironmentFile` / `LoadCredential` — do not `owner = "desk-api"` on a user that does not exist at decrypt time.

Case 3: Postgres on `db-01`

```
# hosts/db-01.nix fragment
{ config, pkgs, ... }:

{
  services.postgresql = {
    enable = true;
    package = pkgs.postgresql_16;
    enableTCPIP = true;
    settings.listen_addresses = "10.100.0.21";
    authentication = pkgs.lib.mkForce ''
      local all all peer
      host  desk_production desk_user 10.100.0.11/32 scram-sha-256
      host  desk_production desk_user 10.100.0.12/32 scram-sha-256
    '';
```

```
    ensureDatabases = [ "desk_production" ];
    ensureUsers = [{
      name = "desk_user";
      ensureDBOwnership = true;
    }];
  };

  # Password is not a Nix string. A oneshot reads sops at runtime (peer as postgres).
  systemd.services.desk-pg-password = {
    after = [ "postgresql.service" ];
    wants = [ "postgresql.service" ];
    wantedBy = [ "multi-user.target" ];
    serviceConfig = {
      Type = "oneshot";
      User = "postgres";
      PrivateTmp = true;
      LoadCredential = "dbpw:${config.sops.secrets.db_password.path}";
      ExecStart = pkgs.writeShellScript "desk-pg-password" ''
        set -euo pipefail
        pw=$(cat "$CREDENTIALS_DIRECTORY/dbpw")
        umask 077
        # Dollar-quote so a password with ' does not break SQL. Not argv.
        printf "ALTER USER desk_user PASSWORD \$\$%s\$\$;\n" "$pw" > /tmp/alter.sql
        ${config.services.postgresql.package}/bin/psql -v ON_ERROR_STOP=1 -f /tmp/alter.sql
        rm -f /tmp/alter.sql
      '';
    };
  };

  # Do not open 5432 on the public NIC. Mesh only.
  networking.firewall.interfaces.wg0.allowedTCPPorts = [ 5432 ];

  environment.persistence."/persist".directories = [ "/var/lib/postgresql" ];
}
```

Do **not** `pkgs.writeText` an `initialScript` that interpolates the password (or even a sops placeholder). `initialScript` is a store path; placeholders only expand in **sops templates**, not in `writeText`. `LoadCredential` + `psql` as `postgres` (peer on the local socket) keeps the bytes out of the NAR. `journalctl` of that oneshot must not use `set -x`.

`listen_addresses = "10.100.0.21"` is the mesh IP of `db-01`, not `*`. Not `postgres` trust on `0.0.0.0/0`.

`ensureUsers` does **not** set a password. `scram-sha-256` without `ALTER USER` is a login that never succeeds. Data dir is `/var/lib/postgresql` — persist it; the store does not hold tables. Dump **before** restic (backups chapter): a running cluster's files without `pg_dumpall` are crash-consistent maybe.

Case 4: Eval the unit path

```
nix eval .#nixosConfigurations.app-01.config.systemd.services.desk-api.serviceConfig.ExecSta
```

Output (shape):

`"/nix/store/…-desk-api-2.0.0/bin/desk-api"`

No `--password=` in that string.

Case 5: Security score

On a deployed app node:

```
systemd-analyze security desk-api --no-pager | tail
curl -fsS http://127.0.0.1:8080/healthz
```

`curl` from the **node** (localhost) or via Caddy. From the public internet, `/healthz` on port 8080 should not answer — only 443 on `gw-01`.

The trap

The trap is **`--password=$DB_PASSWORD` on `ExecStart`**. `ps` shows it. `EnvironmentFile` from sops.

The other trap is `DynamicUser` plus a secret `owner = "desk-api"` that does not exist at decrypt time. For DynamicUser, leave the secret root-owned `0400` and `LoadCredential=` / group, or use a **named** user instead of DynamicUser when the secret must be that uid.

A third: **production Let's Encrypt on the first lab boot** (`acmeCA` left at the public CA while DNS still points at a laptop). Staging first. A fourth: not persisting `/var/lib/caddy` on tmpfs root — new cert every reboot, then a rate limit. A fifth: `listen_addresses = "*"` on postgres.

The boring rule

- TLS only on Caddy. Staging `acmeCA` in lab; `null` in prod. Persist `/var/lib/caddy`.
- Apps HTTP on `wg0`. `buildGoModule` + `CGO_ENABLED=0` + sandbox.
- DB: `listen_addresses` = mesh IP, `pg_hba` `/32`s, 5432 on `wg0` only. Persist the data dir. Dump before restic.
- Secrets: sops file / `LoadCredential`, not argv. `ensureUsers` is not a password.
- Health checks from Caddy over WireGuard.

Try this

1. `systemd-analyze security desk-api` on a lab VM with Case 2.
2. `grep -R password /nix/store/…-desk-api*` should miss the production password (rotate if it hits).
3. From a third VM not on the mesh, `curl` the public IP `:8080` — expect fail; `:443` — expect Caddy.
4. Fail a healthz on `app-01`; Caddy should still serve `app-02`.
5. `nix eval .#nixosConfigurations.gw-01.config.services.caddy.acmeCA` in the lab — staging URL. After cutover, `null`.
6. `ss -lntp | rg 5432` on `db-01` — only `10.100.0.21`.

Automated Fleet Rollout and Disaster Recovery

Automated Fleet Rollout and Disaster Recovery

A production architecture is only as dependable as its playbook. The boring default is: **Colmena canary (`--on app-01`), health probe, then tagged apply; rebuild a dead box with `nixos-anywhere` from the same 26.05 flake — not a restore CD**.

Mental model

The flake is the cluster. Colmena pushes closures. Canary is one host, then a tag. Disaster recovery is Disko + `nixos-anywhere` + restic of `/persist` (not of `/nix/store`).

```
git  →  colmena apply --on app-01  →  curl /healthz
     →  colmena apply --on @app
     →  serial db-01 after restic snapshot
```

Public IPs in the hive are the SSH targets. East-west traffic stays on `wg0` (`10.100.0.0/24`). Rollback is the previous generation, then a git revert so the hive matches metal.

Worked examples

Case 1: Hive for the five-host fleet

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk production fleet";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    colmena = {
      meta.nixpkgs = import nixpkgs { system = "x86_64-linux"; };

      defaults = { ... }: {
        system.stateVersion = "26.05";
        deployment.targetUser = "root";
      };
```

```
      gw-01 = {
        deployment.targetHost = "203.0.113.10";
        deployment.tags = [ "edge" ];
        imports = [ ./hosts/gw-01.nix ];
      };
      app-01 = {
        deployment.targetHost = "203.0.113.11";
        deployment.tags = [ "app" ];
        imports = [ ./hosts/app-01.nix ];
      };
      app-02 = {
        deployment.targetHost = "203.0.113.12";
        deployment.tags = [ "app" ];
        imports = [ ./hosts/app-02.nix ];
      };
      db-01 = {
        deployment.targetHost = "203.0.113.21";
        deployment.tags = [ "db" ];
        imports = [ ./hosts/db-01.nix ];
      };
      ops-01 = {
        deployment.targetHost = "203.0.113.30";
        deployment.tags = [ "ops" ];
        imports = [ ./hosts/ops-01.nix ];
      };
    };
  };
}
```

```
nix shell nixpkgs#colmena --command colmena eval
```

Five nodes. One 26.05 lock. `follows` on disko/sops if those inputs exist. If `colmena eval` says the flake attr moved, the Colmena pin wants `colmenaHive` + `colmena.lib.makeHive` — same hosts, different wrapper (see the Colmena chapter). `deployment.allowLocalDeployment = true` on `ops-01` if that box applies itself.

Case 2: Canary, probe, then the tag

```
colmena apply --dry-run --on app-01
colmena apply --on app-01
curl -sf https://api.desk.internal/healthz
colmena apply --on @app
```

```
[app-01] Built /nix/store/7q1a…-nixos-system-app-01-26.05
[app-01] Activation successful
```

```
HTTP/2 200
```

Non-200: do not apply `@app`. Roll back `app-01` (`nixos-rebuild switch --rollback` on the host, or Colmena's rollback flag), revert git.

Never `--on @all` for a kernel bump. `db-01` is serial, after `restic snapshots` shows a fresh copy.

Case 3: Zero-touch replace of `app-02`

Boot replacement metal into rescue (or a NixOS installer ISO). Disko's `passwordFile` is a path **on the installer** after kexec. Copy the LUKS key there, generate hardware config into git, and do **not** reboot stateful boxes until `/persist` is restored.

From `ops-01`:

```
# LUKS: remote dest (Disko passwordFile) then local source
nix run github:nix-community/nixos-anywhere -- \
  --flake .#app-02 \
  --disk-encryption-keys /tmp/luks.key /run/secrets/luks/app-02 \
  --generate-hardware-config nixos-generate-config ./hosts/app-02/hardware-configuration.nix
  --extra-files /tmp/app-02-seed \
  --no-reboot \
  root@203.0.113.99
```

```
[nixos-anywhere] Connected to rescue over SSH.
[nixos-anywhere] kexec NixOS installer...
[nixos-anywhere] Copied disk encryption key → /tmp/luks.key
[nixos-anywhere] Formatting disks via Disko...
[nixos-anywhere] Copying closure …-nixos-system-app-02-26.05
[nixos-anywhere] Copied extra files onto the new root
[nixos-anywhere] Installation complete. (--no-reboot)
```

`--disk-encryption-keys <remote> <local>` runs after kexec, before Disko. Repeat the flag per disk. The key is **not** in the flake.

`--generate-hardware-config nixos-generate-config <path>` writes initrd modules and NIC names back to the laptop so the next commit has a real hardware file. Backend can be `nixos-facter` instead; pick one.

`--extra-files <dir>` is a **tree rooted at `/`** of the new install (`etc/ssh/ssh_host_ed25519_key`, `persist/…`). Prepare it with `cp --parents`. This is how sops still decrypts on first boot (same host key). `--copy-host-keys` only copies keys **from the machine you are wiping**, which a blank replacement does not have.

`--no-reboot` leaves disks mounted so Case 4 can restore. Default phases are `kexec,disko,install,reboot`. Already on a NixOS installer ISO: `--phases disko,install` (skip kexec). `--vm-test` exercises Disko without touching metal.

Then reboot (or omit `--no-reboot` on **stateless** `app-*`). The node joins WireGuard (`10.100.0.12`). Ephemeral workers have nothing to restic. Stateful `db-01` needs a snapshot **before** you wipe, and a restore **before** you reboot into postgres.

Case 4: Restore `/persist` on `db-01` (not the store)

```
# still in the installer (or chroot) after --no-reboot, before first Stage 2 postgres:
sudo restic -r s3:s3.amazonaws.com/desk-restic snapshots
sudo restic -r s3:s3.amazonaws.com/desk-restic restore latest --target /mnt/persist
# then reboot into the new generation
sudo umount -R /mnt
sudo reboot
```

After boot:

```
sudo systemctl start postgresql
sudo -u postgres pg_isready
```

`--target /persist` vs `/mnt/persist` depends on whether you have already switched root. Wrong prefix dumps the tree next to the subvolume and postgres starts empty. The restic password is sops on the box (or `RESTIC_PASSWORD_FILE`), not argv. `/nix/store` comes from the flake + cache. Restoring a NAR of the old store is not DR.

If you omitted `--no-reboot`, postgres may already have `initdb`'d an empty cluster on the new persist. Restore **onto** that is a fight. Stop postgres, empty the data dir, restore, start. Better: never let it boot empty.

Case 5: Quarterly drill is the playbook

Save as `docs/dr-drill.md`:

```
# docs/dr-drill.md
1. Snapshot restic on db-01.
2. Wipe staging-app in the lab (not prod).
3. nixos-anywhere --flake .#staging-app root@…
4. curl /healthz from gw.
5. Time it. File the minutes in this repo.
```

If the drill needs a human in a web console past step 3, the ISO / Disko / sops recipients are incomplete. Fix the flake, not the wiki.

The trap

The trap is **not testing disaster recovery until a production incident**. The other trap is applying `db-01` in the same parallel wave as `@app` while a migration runs. Serial the stateful node.

Canary the stateless ones.

Rollback without a git revert means the next Colmena apply re-breaks the box.

A third trap is **rebooting `nixos-anywhere` before `/persist` is restored** on a stateful host — or omitting `--disk-encryption-keys` so Disko waits on a passphrase nobody will type in rescue. A fourth is generating hardware config and never committing it: the next install misses a kernel module.

The boring rule

- One hive, one 26.05 lock. SSH from `ops-01`.
- `--dry-run`, `--on app-01`, healthz, then `--on @app`. Serial `db-01`.
- Dead metal: `nixos-anywhere` + Disko. `--disk-encryption-keys`, `--generate-hardware-config`, `--extra-files` for host keys. `--no-reboot` until persist is restored on stateful nodes.
- Persist: restic. Store: substituters.
- Rollback is a generation, then git revert.
- DR drill in git. Time it.

Try this

1. `colmena eval` and list tags.
2. Dry-run `--on app-01`; read which units would restart.
3. Write the restic repository URL (not the password) next to the drill doc.
4. Wipe a lab VM and `nixos-anywhere` it with `--no-reboot`; restore a dummy file into `/mnt/persist`; reboot; confirm it survived. `wg show` has peers.

Production Labs

Reading NixOS.org Infra as a Pattern Source

Reading NixOS.org Infra as a Pattern Source

The public tree at `github:NixOS/infra` is how the NixOS project runs hydra, ofborg, builders, DNS, and terraform. The boring default is: **steal the *shapes* (keys file, host factory, critical/non-critical split, restricted builder SSH, channel table) and re-express them on the desk fleet — do not clone the repo and do not deploy their hosts**.

This part is labs. The capstone already has five desk roles. Here we add the org-scale habits that tree actually uses in 2026.

Mental model

Path in NixOS/infra	What it is	Desk equivalent
`flake.nix`	Live contract: 26.05-small, `follows`, extra substituter	One lock, one cache
`keys.nix`	People and machines as SSH/age groups	`keys.nix` for desk-ops
`builders/`	`mkNixOS` factory + common modules	`lib/mkHost.nix`
`non-critical-infra/`	Separate Colmena hive + sops	`hive-apps` vs `hive-core`
`channels.nix`	Hydra job → channel name + status	Attic/Cachix "tested" attr
`terraform/`	OpenTofu `withPlugins` for the cloud envelope	IaC chapters, same wrapper
`checks/`	Group host toplevels by arch for CI	`checks.x86_64-linux.hosts`
`docs/inventory.md`	**Stale** (NixOps, Packet, DataDog)	Do not copy; flake wins

The README still says "managed using NixOps." The flake says Colmena, Disko, sops-nix, agenix, nix-darwin 26.05, OpenTofu. **Trust the flake.** Inventory docs rot; `flake.lock` does not.

They use **flake-parts**. We do not. The factory is a function in `flake.nix`. Same hosts, no extra DSL.

They pin `https://channels.nixos.org/nixos-26.05-small/nixexprs.tar.zst` (the *small* channel). The desk default stays `github:NixOS/nixpkgs/nixos-26.05` unless you have measured that you only need the small jobset.

```
NixOS/infra (read)
    keys, factory, split hives, restricted SSH, channel table
            rewrite

desk flake (write)  -  26.05, Colmena, sops, Disko
```

Worked examples

Case 1: Map the live flake, ignore the stale README

On a throwaway clone (read-only; you will not `colmena apply`):

```
git clone --depth 1 https://github.com/NixOS/infra.git /tmp/nixos-infra
cd /tmp/nixos-infra
rg -n 'nixos-26.05|colmena|nixops|disko|sops' flake.nix README.md | head
```

Output (shape):

```
flake.nix:    nixpkgs.url = "https://channels.nixos.org/nixos-26.05-small/nixexprs.tar.zst";
flake.nix:    colmena = { url = "github:zhaofengli/colmena"; ...
flake.nix:    disko = { url = "github:nix-community/disko"; ...
flake.nix:    sops-nix = { url = "github:Mic92/sops-nix"; ...
README.md:All the hosts are currently managed using NixOps.
```

README lost. Flake won. That is the first lab skill: **the deploy tool is the flake output, not the first paragraph of the README**.

```
nix flake metadata --json 2>/dev/null | jq -r '.locks.nodes.nixpkgs.original.url // .locks.n
```

You should see a 26.05-small tarball or its `narHash`, not `nixpkgs-unstable` as the default OS pin. Unstable is a **second** input (`nixpkgs-unstable`) for tools that need it.

Case 2: Extra substituter on the flake, daemon still needs the key

They set:

```
# flake.nix fragment - pattern only
{
  nixConfig.extra-substituters = [ "https://nixos-infra-dev.cachix.org" ];
  nixConfig.extra-trusted-public-keys = [
    "nixos-infra-dev.cachix.org-1:OvwhqPPs81cInrtRAX0K7dG6lw8wXcQEX4xyp4AnSXw="
  ];
}
```

`nixConfig` in a flake is a **hint**. Untrusted users need `accept-flake-config` or the same keys in `/etc/nix/nix.conf`. The desk already puts the team cache in `nix.settings`. Do not copy their Cachix name onto the desk fleet; copy the *habit*: one extra cache, key next to URL, never a signing secret in `nixConfig`.

Case 3: `follows` is how a 20-input flake stays on one nixpkgs

Their `colmena`, `disko`, `sops-nix`, `srvos`, `agenix` all `inputs.nixpkgs.follows = "nixpkgs"`. Darwin follows `nixpkgs-darwin`. That is the same `follows` rule as the tooling-flake chapter, at org scale.

```
# in the clone
rg 'follows = "nixpkgs"' flake.nix | wc -l
```

A two-digit count is success. An input that does **not** follow is a second stdenv. Hunt those on purpose (hydra, ofborg) — they are the exceptions, documented by existing.

Case 4: Desk flake that *looks* like infra without flake-parts

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk fleet - shapes from NixOS/infra, not their hosts";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    disko.url = "github:nix-community/disko";
    disko.inputs.nixpkgs.follows = "nixpkgs";
    sops-nix.url = "github:Mic92/sops-nix";
    sops-nix.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, disko, sops-nix }:
    let
      mkHost = name: extraModules:
        nixpkgs.lib.nixosSystem {
          system = "x86_64-linux";
          modules = [
            { networking.hostName = name; system.stateVersion = "26.05"; }
            disko.nixosModules.disko
            sops-nix.nixosModules.sops
          ] ++ extraModules;
        };
    in
    {
```

```
      nixosConfigurations = {
        gw-01  = mkHost "gw-01"  [ ./hosts/gw-01.nix ];
        app-01 = mkHost "app-01" [ ./hosts/app-01.nix ];
        ops-01 = mkHost "ops-01" [ ./hosts/ops-01.nix ];
      };
    };
}
```

`mkHost` is their `mkNixOS` without flake-parts. Three hosts are enough for the labs in this part. Capstone still has five.

```
nix flake show
```

```
nixosConfigurations
    app-01
    gw-01
    ops-01
```

Case 5: What we will not copy

NixOS/infra	Why not on the desk
Hydra of all of nixpkgs	A queue-runner for the world. Desk CI is `nix flake check`.
ofborg eval fleet	GitHub PR eval for nixpkgs. Not your app.
`srvos` as a required input	Fine later; not the first module.
flake-parts	Extra DSL. A function is enough.
Their `keys.nix` values	Public SSH keys of *their* operators. Never paste into yours.
`isoImage.contents` of `key.txt`	Private keys stay off git.
NixOps	The README; not the flake.

Open `keys.nix` in the clone, **look at the shape** (`ssh.users`, `ssh.groups`, `ssh.machines`, `age.recipients`), then close it. Next chapter you write *your* keys.

The trap

The trap is **`git clone` + `colmena apply` against `staging-hydra.nixos.org`**. That hive is not yours. Read-only clone. Rewrite patterns into the desk flake.

The other trap is treating `docs/inventory.md` as current. Packet.net, NixOps, and DataDog in that file are archaeology. The flake and `non-critical-infra/` are the live map.

The boring rule

- Flake is the contract. README and inventory can lie.
- Steal shapes: keys file, host factory, hive split, restricted builder SSH, channel table.
- Do not steal hosts, secrets, flake-parts, or Hydra-for-nixpkgs.
- `follows` on every input that can. Extra substituter + key, never a signing secret.
- Desk pin stays `nixos-26.05` unless you have a reason for `-small`.

Try this

1. Clone NixOS/infra at depth 1. Diff `README.md` vs `flake.nix` for the deploy tool.
2. Count `follows = "nixpkgs"` in `flake.nix`.
3. `rg NixOps` the repo; note which files still say it.
4. Write `mkHost` as in Case 4 in a throwaway dir; `nix flake show`. Do not add their Cachix.

Operator Keys and Groups

Operator Keys and Groups

NixOS/infra keeps every operator and machine pubkey in one `keys.nix`: users, groups (`infra-core`, `infra`, `ofborg`), machine host keys, age recipients. The boring default is: **the same file for the desk — public keys only — and `users.mutableUsers = false` so a leftover `authorized_keys` on disk cannot add a human**.

Do not copy their key strings. Copy the attrset.

Mental model

```
keys.nix
  ssh.users.alice    = [ "ssh-ed25519 AAAA… alice@laptop" ]
  ssh.groups.desk-core = alice ++ bob
  ssh.machines.gw-01 = "ssh-ed25519 AAAA…"   # host key, for knownHosts
  age.recipients.alice = [ "age1…" ]


users.users.root.openssh.authorizedKeys.keys = keys.ssh.groups.desk-core;
```

Groups are concatenations of user lists. Adding alice to `desk-core` is one line; every host that imports the group gets her key on the next switch. Removing her is the same line, plus a rebuild — not `sed` on five boxes.

`mutableUsers = false` means NixOS owns `/etc/passwd` and authorized keys. A `useradd` on the box dies at next activation.

Age groups in their file mix SSH groups with YubiKey `age1yubikey1…` recipients. Desk: sops `.sops.yaml` creation rules point at the same people. One list of humans.

Worked examples

Case 1: Desk `keys.nix`

Save as `keys.nix`:

```nix
# keys.nix
rec {
  ssh = {
    users = {
```

```
      alice = [
        "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskAliceExampleKeyOnlyNotReal alice@laptop"
      ];
      bob = [
        "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskBobExampleKeyOnlyNotReal bob@laptop"
      ];
      hydra-queue = [
        "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskHydraExampleKeyOnlyNotReal hydra-queue@ops
      ];
    };

    groups = with ssh.users; {
      desk-core = alice ++ bob;
      desk-builders = desk-core;
      desk-ci = desk-core ++ hydra-queue;
    };

    machines = {
      gw-01 = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskGw01HostKeyExampleOnly";
      app-01 = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskApp01HostKeyExampleOnly";
      ops-01 = "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIDeskOps01HostKeyExampleOnly";
    };
  };

  age = {
    recipients = {
      alice = [ "age1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq2w4k7f" ];
    };
    groups = {
      desk-core = ssh.groups.desk-core;
    };
  };
}
```

Replace every `AAAA…` / `age1…` with **your** pubkeys (`ssh-keygen -y`, `ssh-to-age < ~/.ssh/id_ed25519.pub`). The strings above are placeholders; they must not be committed as if they opened a box.

```
nix eval --file keys.nix 'ssh.groups.desk-core' --apply builtins.length
```

```
2
```

Case 2: Root and a named admin from the group

Save as `modules/users-from-keys.nix`:

```
# modules/users-from-keys.nix
{ pkgs, ... }:

let
  keys = import ../keys.nix;
in
{
  users.mutableUsers = false;

  users.users.root.openssh.authorizedKeys.keys = keys.ssh.groups.desk-core;

  users.users.deskadmin = {
    isNormalUser = true;
    extraGroups = [ "wheel" ];
    openssh.authorizedKeys.keys = keys.ssh.groups.desk-core;
  };

  security.sudo.wheelNeedsPassword = false; # lab; prod: password or SSH-only + sudoers
  services.openssh = {
    enable = true;
    settings = {
      PasswordAuthentication = false;
      KbdInteractiveAuthentication = false;
      PermitRootLogin = "prohibit-password";
    };
  };
}
```

NixOS/infra uses `users.extraUsers.root` (legacy alias). On 26.05 write `users.users.root`. Same option.

```
nix eval .#nixosConfigurations.ops-01.config.users.users.root.openssh.authorizedKeys.keys --
```

Must equal the length of `desk-core`.

Case 3: `knownHosts` from `ssh.machines`

Save as `modules/known-hosts.nix`:

```
# modules/known-hosts.nix
{
  programs.ssh.knownHosts = let
    keys = import ../keys.nix;
  in
    builtins.mapAttrs (_name: publicKey: { inherit publicKey; }) keys.ssh.machines;
}
```

Colmena/SSH from `ops-01` then does not prompt on first connect. Rotate a host key → change `keys.nix` → switch `ops-01`. Leaving `StrictHostKeyChecking=no` in Colmena is how you accept a MITM once.

Host keys on ephemeral roots must be **persisted** (`/etc/ssh/ssh_host_ed25519_key`) or this file and the metal disagree every boot.

Case 4: Drop a human in one commit

```
# keys.nix fragment after alice leaves
groups = with ssh.users; {
  desk-core = bob;          # alice removed
  desk-ci = bob ++ hydra-queue;
};
```

```
nix eval --file keys.nix 'ssh.groups.desk-core' --apply 'xs: builtins.any (k: builtins.match
```

```
false
```

Then `colmena apply --on @core`. There is no "disable the account on the bastion wiki." The next generation has no alice key. Rotate anything she could have copied (sops recipients, age, Cachix tokens).

Case 5: sops creation rules share the same people

Save as `.sops.yaml` (desk, not their file):

```
# .sops.yaml
keys:
  - &alice age1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq2w4k7f
  - &bob   age1qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqp9z7n0h
creation_rules:
  - path_regex: secrets/core/.*
    key_groups:
      - age: [*alice, *bob]
  - path_regex: secrets/apps/.*
    key_groups:
      - age: [*alice, *bob]
```

Need-to-have: `secrets/apps/` does not need a laptop that only ships CI. NixOS/infra's non-critical README says the same: add your key on a PR, someone with access runs `sops updatekeys`. Desk: PR to `keys.nix` **and** `.sops.yaml`, then `sops updatekeys secrets/core/*.yaml`.

The trap

The trap is **pasting NixOS/infra `keys.nix` into the desk repo**. Those keys open *their* machines. Generate yours. The other trap is `mutableUsers = true` plus this file: a human `ssh-copy-id`s a third key and Nix never removes it.

A third: listing alice in `desk-core` but forgetting `.sops.yaml` — she can SSH and cannot decrypt. Or the reverse: she decrypts on the laptop and cannot log in.

The boring rule

- One `keys.nix`. Public keys only. Groups are concatenations.
- `mutableUsers = false`. Authorized keys come from the group.
- `knownHosts` from `ssh.machines`. Persist host keys on tmpfs roots.
- Drop a human in `keys.nix` + sops recipients + apply. Then rotate.
- Never commit private keys. Never copy another org's pubkeys as if they were yours.

Try this

1. Put two of *your* pubkeys in `keys.nix`. `nix eval` the group length.
2. Import `users-from-keys.nix` on a lab VM; `ssh -i` as each key; remove one; switch; confirm the removed key is refused.
3. `ssh-keyscan` a lab host; put it in `ssh.machines`; rebuild `ops-01`; `ssh` without a prompt.
4. `git grep -n 'ssh-ed25519' keys.nix` — every line is a key you can name.

Critical vs Non-Critical Hives

Critical vs Non-Critical Hives

NixOS/infra splits **builders / hydra / ofborg** from `non-critical-infra/` (community hosts, staging hydra, a second Colmena hive, its own `.sops.yaml` and `devShells.non-critical-infra`). The boring default is: **two Colmena tags (or two flakes) so a botched app host cannot share sops keys or apply waves with `gw-01`.**

The capstone hive is one file. This lab splits it the way a real org does once "the website" and "the CA / builders" must not share a blast radius.

Mental model

```
flake.nix
  colmena.meta
  defaults            →  ssh keys, firewall, stateVersion
  @core  gw-01 db-01 ops-01     secrets/core/
  @apps  app-01 app-02          secrets/apps/
```

NixOS/infra goes further: a **directory** with its own sops and `colmena.sh`. Desk: one flake, two tags, two secret trees. Two flakes when the team that owns apps must not eval the core modules.

Their non-critical README: secrets are need-to-have; add your key on a PR; someone who already has access runs `updatekeys`. That is the onboarding ritual. Not a Slack DM of `key.txt`.

Worked examples

Case 1: Tags on the existing hive

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk hive with core vs apps";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    colmena = {
      meta.nixpkgs = import nixpkgs { system = "x86_64-linux"; };
```

```
      defaults = { ... }: {
        system.stateVersion = "26.05";
        deployment.targetUser = "root";
        imports = [ ./modules/users-from-keys.nix ];
      };

      gw-01 = {
        deployment.targetHost = "203.0.113.10";
        deployment.tags = [ "core" "edge" ];
        imports = [ ./hosts/gw-01.nix ];
      };
      db-01 = {
        deployment.targetHost = "203.0.113.21";
        deployment.tags = [ "core" "db" ];
        imports = [ ./hosts/db-01.nix ];
      };
      ops-01 = {
        deployment.targetHost = "203.0.113.30";
        deployment.tags = [ "core" "ops" ];
        imports = [ ./hosts/ops-01.nix ];
      };
      app-01 = {
        deployment.targetHost = "203.0.113.11";
        deployment.tags = [ "apps" ];
        imports = [ ./hosts/app-01.nix ];
      };
      app-02 = {
        deployment.targetHost = "203.0.113.12";
        deployment.tags = [ "apps" ];
        imports = [ ./hosts/app-02.nix ];
      };
    };
  };
}
```

```
nix shell nixpkgs#colmena --command colmena eval -E '{ nodes }: builtins.attrNames nodes'
```

Five names. Apply waves:

```
colmena apply --on @core --dry-run
colmena apply --on @apps --dry-run
```

Never `--on @all` for a kernel bump on `@core`. Apps can fan out. `db-01` stays serial inside `@core` (capstone DR).

Case 2: Two sops trees

```
secrets/
  core/
    luks.yaml          # recipients: desk-core only
    restic.yaml
  apps/
    db-password.yaml   # recipients: desk-core + app runtime? no - hosts decrypt
```

App hosts decrypt `db-password` with the **host** age key, not alice's laptop. Alice's key is for editing the YAML. That is the same split as NixOS/infra: humans in `.sops.yaml`, machines as extra recipients.

Save as `modules/sops-core.nix`:

```
# modules/sops-core.nix
{ config, ... }:
{
  sops.defaultSopsFile = ../secrets/core/luks.yaml;
  sops.age.sshKeyPaths = [ "/etc/ssh/ssh_host_ed25519_key" ];
}
```

Save as `modules/sops-apps.nix`:

```
# modules/sops-apps.nix
{
  sops.defaultSopsFile = ../secrets/apps/db-password.yaml;
  sops.age.sshKeyPaths = [ "/etc/ssh/ssh_host_ed25519_key" ];
}
```

Import `sops-core.nix` only on `@core`. Import `sops-apps.nix` on `@apps` (and `db-01` if the DB password lives there). An app box that can decrypt LUKS material is the wrong blast radius.

Case 3: Named devShells, not one kitchen sink

NixOS/infra exposes `devShells.builders` (agenix) and `devShells.non-critical-infra` (colmena, sops, ssh-to-age) and `devShells.terraform` (OpenTofu `withPlugins`). Desk:

```
# flake.nix fragment
{
  devShells.x86_64-linux.core = pkgs.mkShell {
    packages = [ pkgs.colmena pkgs.sops pkgs.ssh-to-age ];
  };
  devShells.x86_64-linux.apps = pkgs.mkShell {
    packages = [ pkgs.colmena pkgs.sops ];
  };
  devShells.x86_64-linux.tf = pkgs.mkShell {
```

```
    packages = [
      (pkgs.opentofu.withPlugins (p: [ p.aws p.local ]))
    ];
  };
}
```

The app team's direnv is `use flake .#apps`. They do not get the OpenTofu wrapper or core sops files in `$PWD` if those live in another worktree. Need-to-have.

Case 4: Staging is a host, not a branch of secrets

Their Colmena list includes `staging-hydra.targetHost = "staging-hydra.nixos.org"` **in the non-critical hive**. A hydra change is exercised there before the production hydra input moves.

Desk equivalent:

```
# hosts/staging-app.nix
{
  imports = [ ./app-01.nix ];
  networking.hostName = "staging-app";
  # acmeCA staging; smaller RAM; same modules
}
```

```
colmena apply --on staging-app
curl -sf https://staging.api.desk.internal/healthz
colmena apply --on @apps
```

Do not "staging" by decrypting prod sops on a laptop and hoping. Same modules, different host, different ACME, different tag.

Case 5: `colmena.sh` is a one-liner, not a platform

NixOS/infra's `non-critical-infra/colmena.sh` is a thin wrapper so humans do not forget `--flake`. Desk:

```
# colmena-apps.sh
#!/usr/bin/env bash
set -euo pipefail
exec nix shell nixpkgs#colmena --command colmena "$@" --on @apps
```

```
chmod +x colmena-apps.sh
./colmena-apps.sh apply --dry-run
```

No extra DSL. The wrapper only *restricts* the tag. A wrapper that defaults to `@all` is worse than none.

The trap

The trap is **one sops file for LUKS, restic, and the app DB password**. A contractor who needs `desk-api` then decrypts disk keys. Split trees. Split tags.

The other trap is applying `@core` and `@apps` in one command because "it's faster." Core is serial and canary. Apps are tagged.

The boring rule

- `@core` vs `@apps` (or two flakes). No `@all` for kernels.
- `secrets/core/` vs `secrets/apps/`. Host age keys decrypt; humans rekey.
- Staging is a host with the same modules.
- Named `devShells` per team. Need-to-have sops.
- A 10-line `colmena-*.sh` that pins the tag is enough.

Try this

1. Add tags to the capstone hive; `colmena eval` and list `@core` vs `@apps`.
2. Move a dummy secret into `secrets/apps/` and confirm a `--dry-run` on `@core` does not need it.
3. Write `colmena-apps.sh`; run `--dry-run`; confirm it refuses to take `--on gw-01` or wrap so that `--on` is ignored.
4. Add `staging-app` as a sixth host; apply it before `@apps`.

Builder Factory and Restricted Store SSH

Builder Factory and Restricted Store SSH

NixOS/infra's `builders/` is a function `mkNixOS system config` that always imports the same common modules (hardening, ssh, nix, hydra-queue-builder, users), then a per-box instance (CPU, disks). The hydra queue runner may SSH in **only** as `nix-store --serve --write` — not a shell. The boring default is: **the same factory for desk builders, and a forced SSH command so CI cannot `rm -rf`**.

The remote-builders chapter listed `nix.buildMachines`. This lab is the **other** side of that SSH: the box that receives the `.drv`.

Mental model

```
ops-01 / hydra-queue
    ssh build@builder-01
         forced command: nix-store --serve --write


         builder-01  (kvm, big-parallel)
                NAR

         team cache
```

`users.users.build` (uid 2000 in their tree) holds **only** that authorized key, wrapped:

```
command="NIX_SSL_CERT_FILE=…/ca-bundle.crt  /nix/store/…-nix/bin/nix-store --serve --write" ssh
```

`authorizedKeys command=` is OpenSSH forced command. Agent forwarding and `pty` will not get you bash. That is the point.

Common modules stay boring: UTC, `mutableUsers = false`, nftables, `openssh`, nix flakes, node-exporter. Instance files set Disko `by-id` and `system`.

Worked examples

Case 1: `mkHost` factory with common modules

Save as `lib/mk-builder.nix`:

```
# lib/mk-builder.nix
{ nixpkgs, disko }:

system: extraModules:
nixpkgs.lib.nixosSystem {
  inherit system;
  modules = [
    disko.nixosModules.disko
    ../builders/common/system.nix
    ../builders/common/ssh.nix
    ../builders/common/nix.nix
    extraModules
  ];
}
```

Save as `builders/common/system.nix`:

```
# builders/common/system.nix
{ pkgs, ... }:
{
  time.timeZone = "UTC";
  users.mutableUsers = false;
  environment.systemPackages = [ pkgs.git pkgs.jq ];
  system.stateVersion = "26.05";
}
```

Save as `flake.nix`:

```
# flake.nix
{
  description = "Desk builders";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    disko.url = "github:nix-community/disko";
    disko.inputs.nixpkgs.follows = "nixpkgs";
  };

  outputs = { self, nixpkgs, disko }:
    let
      mkBuilder = import ./lib/mk-builder.nix { inherit nixpkgs disko; };
    in
    {
      nixosConfigurations = {
        builder-01 = mkBuilder "x86_64-linux" [ ./builders/instances/builder-01.nix ];
        builder-arm = mkBuilder "aarch64-linux" [ ./builders/instances/builder-arm.nix ];
      };
```

```
    };
}
```

Two archs, one factory. NixOS/infra's comments list cores and RAM next to each instance — do that in a comment, not in a wiki.

Case 2: Forced `nix-store --serve --write`

Save as `builders/common/ssh.nix`:

```
# builders/common/ssh.nix
{ config, lib, pkgs, ... }:

let
  keys = import ../../keys.nix;
  authorizedNixStoreKey = key:
    let
      env = "NIX_SSL_CERT_FILE=${pkgs.cacert}/etc/ssl/certs/ca-bundle.crt";
      nixStore = "${config.nix.package}/bin/nix-store";
    in
    ''command="${env} ${nixStore} --serve --write" ${key}'';
in
{
  services.openssh.enable = true;
  services.openssh.settings.PasswordAuthentication = false;
  services.openssh.settings.KbdInteractiveAuthentication = false;

  users.users.root.openssh.authorizedKeys.keys = keys.ssh.groups.desk-core;

  users.users.build = {
    isNormalUser = true;
    uid = 2000;
    # no extraGroups. no lingering.
    openssh.authorizedKeys.keys =
      map authorizedNixStoreKey keys.ssh.users.hydra-queue;
  };

  nix.settings.trusted-users = [ "root" "build" ];
}
```

`trusted-users` is required for `--serve --write` to import paths. Keep it `root` + `build`, not `@wheel`.

```
nix eval .#nixosConfigurations.builder-01.config.users.users.build.openssh.authorizedKeys.ke
```

Output (shape):

`command="NIX_SSL_CERT_FILE=/nix/store/…`

If the line is a bare `ssh-ed25519`, the force is missing and CI has a shell.

Case 3: Nix daemon knobs the factory always sets

Save as `builders/common/nix.nix`:

```
# builders/common/nix.nix
{
  nix.settings = {
    experimental-features = [ "nix-command" "flakes" ];
    cores = 0;           # one job may use all CPUs; pair with max-jobs
    max-jobs = 8;        # set from nproc / RAM on the instance
    sandbox = true;
    builders-use-substitutes = true;
    extra-substituters = [ "https://cache.nixos.org" ];
    extra-trusted-public-keys = [
      "cache.nixos.org-1:6NCHdD59X431o0gWypbMrAURkbJ16ZPMQFGspcDShjY="
    ];
  };
  nix.gc.automatic = true;
  nix.gc.dates = "weekly";
  nix.gc.options = "--delete-older-than 7d";
  boot.binfmt.emulatedSystems = [ ]; # add aarch64-linux only if you meant qemu
}
```

NixOS/infra sets `cores = 0` in `modules/common.nix`. On a shared builder, `max-jobs` is the throttle; `cores = 0` lets a single Chromium use the box. Do not copy `max-jobs = 96` from their Epyc comment onto a 4-core lab VM.

Case 4: Client `buildMachines` matches the forced user

Save as `modules/use-builder-01.nix` on `ops-01`:

```
# modules/use-builder-01.nix
{
  nix.buildMachines = [{
    hostName = "builder-01.desk.internal";
    sshUser = "build";
    sshKey = "/run/secrets/ops-builder-ssh";
    system = "x86_64-linux";
    protocol = "ssh-ng";
    maxJobs = 8;
    speedFactor = 4;
    supportedFeatures = [ "kvm" "nixos-test" "big-parallel" ];
```

```
    }];
    nix.distributedBuilds = true;
    nix.settings.builders-use-substitutes = true;
  }
```

The key in `sshKey` is the **private** half of `keys.ssh.users.hydra-queue`, on sops, mode `0400`. The public half is what Case 2 wrapped in `command=`. If you use alice's laptop key here, alice can `--serve --write` **and** she still has a root shell from `desk-core` — two roles, two keys.

```
nix build .#nixosConfigurations.app-01.config.system.build.toplevel --max-jobs 0
```

`--max-jobs 0` forbids a local compile. The drv must go to `builder-01`. If it builds on `ops-01`, the feature flags or SSH forced command are wrong (`journalctl` on the builder, `ssh -v build@builder-01`).

Case 5: Instance file is disks + identity, not a second OS

Save as `builders/instances/builder-01.nix`:

```
# builders/instances/builder-01.nix
{ lib, ... }:
{
  networking.hostName = "builder-01";
  # Epyc-class in prod; lab: one disk.
  disko.devices.disk.main.device = lib.mkDefault "/dev/vda";
  imports = [ ../disk-layouts/ext4.nix ];

  nix.settings.max-jobs = 8;
  nix.settings.system-features = [ "kvm" "nixos-test" "big-parallel" ];
}
```

Hardware comments belong here ("48C/96T, 256 GB") so the next human does not schedule Chromium on a cloud burstable. The OS is the factory. The instance is numbers.

The trap

The trap is **a builder account with a login shell and `wheel`**. The queue runner then has root. Forced `command=` + `trusted-users = [ "build" ]` is the whole privilege.

The other trap is wrapping `command=` around **root**'s keys so you cannot SSH in to repair. Root stays `desk-core` without a force. `build` is the restricted user.

A third: `boot.binfmt.emulatedSystems = [ "aarch64-linux" ]` on an x86 builder so "we don't need builder-arm." Qemu is not a 96-thread Ampere. Separate `system` in the factory.

The boring rule

- One `mkBuilder` function. Common modules always. Instance = disks + jobs + features.
- `build` user: forced `nix-store --serve --write`, `trusted-users`, no `wheel`.
- Root: `desk-core` keys, no force.
- Client `sshUser = "build"` + sops private key. `--max-jobs 0` to prove it.
- `system-features` advertised must match what you put in `supportedFeatures` on the client.

Try this

1. Eval Case 2; confirm the `build` key starts with `command="`.
2. On a lab VM, add the `build` user; `ssh build@vm` — must not get a shell; `nix store ping --store ssh-ng://build@vm` (or `nix ping-store`) from a client that has the matching key.
3. `nix build nixpkgs#hello --max-jobs 0` from `ops-01` with Case 4; `journalctl -u nix-daemon` on the builder shows the compile (or a substitute).
4. Add `builder-arm` with `system = "aarch64-linux"`; `nix flake show`. Do not binfmt it onto builder-01 for the lab.

Channel Promotion and CI Host Groups

Channel Promotion and CI Host Groups

NixOS/infra's `channels.nix` is a table: channel name → Hydra job, variant (`primary` / `small` / `darwin`), status (`rolling` / `stable` / `unmaintained`). Their `checks/` flake groups host **toplevels** by architecture so CI can `nix-fast-build` one arch without evaluating every Darwin Mac. The boring default is: **the same two ideas at desk scale — a promotion table in git, and `checks` that list host closures by arch — not a wiki "we shipped 26.05."**

You will not run Hydra for nixpkgs. You *will* have a `tested` job that may update a Cachix/Attic alias.

Mental model

```
nix flake check          →  hosts eval + tests
nix build .#tested       →  the closure you are willing to name
cachix push / attic push →  cache
channels.nix status      →  stable | staging | unmaintained
```

NixOS maps `nixos-26.05` to `nixos/release-26.05/tested` on hydra.nixos.org. Desk maps `desk-stable` to `.#nixosConfigurations.app-01.config.system.build.toplevel` **after** `@apps` canary. The table is data. The job is CI.

`25.11` in their file is `status = "unmaintained"`. That is how you tell humans to stop following a pin — not a Slack message.

CI groups (their `flake.ciSystems.ofborg-x86_64-linux = [ core01 build01 … ]`) exist so one runner does not eval aarch64-darwin. List names **explicitly**. Do not `mapAttrs` every `nixosConfigurations` if that forces Mac eval on a Linux runner.

Worked examples

Case 1: A desk channel table

Save as `channels.nix`:

```
# channels.nix
{
  channels = {
    desk-unstable = {
      job = "flake.checks.x86_64-linux.default";
```

```
      variant = "primary";
      status = "rolling";
    };
    desk-26.05 = {
      job = "nixosConfigurations.app-01.toplevel";
      variant = "primary";
      status = "stable";
    };
    desk-26.05-small = {
      job = "nixosConfigurations.app-01.toplevel";
      variant = "small";
      status = "stable";
    };
  };
}
```

```
nix eval --file channels.nix 'channels.desk-26.05.status'
```

```
"stable"
```

When 26.11 exists, set `desk-26.05.status = "deprecated"` in the same PR that bumps the flake input. The table is the announcement.

Case 2: `tested` as an explicit package

Save as `flake.nix` fragment:

```
# flake.nix fragment
{
  packages.x86_64-linux.tested = pkgs.releaseTools.aggregate {
    name = "desk-tested";
    constituents = [
      self.nixosConfigurations.app-01.config.system.build.toplevel
      self.nixosConfigurations.app-02.config.system.build.toplevel
      self.checks.x86_64-linux.fmt
    ];
  };
}
```

If `releaseTools.aggregate` feels heavy, a simpler pin:

```
# flake.nix fragment
{
  packages.x86_64-linux.tested =
    self.nixosConfigurations.app-01.config.system.build.toplevel;
}
```

CI:

```
nix build .#tested
cachix push desk-cache ./result
```

Humans `nixos-rebuild switch --flake .#app-01` only after `.tested` is green. That is a channel. Hydra's UI is optional.

Case 3: Group host toplevels by arch (no surprise Darwin eval)

Save as `ci-hosts.nix`:

```
# ci-hosts.nix
{ self, lib }:

let
  nixos = names:
    lib.genAttrs names (n: self.nixosConfigurations.${n}.config.system.build.toplevel);
in
{
  desk-x86_64-linux = nixos [ "gw-01" "app-01" "app-02" "db-01" "ops-01" ];
  desk-aarch64-linux = nixos [ "builder-arm" ];
}
```

In `outputs`:

```
# flake.nix fragment
{
  checks.x86_64-linux = (import ./ci-hosts.nix { inherit self; lib = nixpkgs.lib; }).desk-x8
}
```

```
nix flake check
```

A Linux GHA runner evals five x86_64 toplevels. It does **not** eval `builder-arm` unless you add a matrix job for `aarch64-linux`. NixOS/infra's comment: list hosts explicitly so CI does not force every configuration just to learn `system`.

Case 4: OpenTofu envelope, NixOS OS — two shells

Their `terraform/flake-module.nix` is a `mkShellNoCC` with `opentofu.withPlugins` (AWS, Fastly, Netlify). DNS and IAM live there. Machines live in Colmena.

```
# flake.nix fragment
{
  devShells.x86_64-linux.tf = pkgs.mkShell {
    packages = [
```

```
      pkgs.awscli2
      (pkgs.opentofu.withPlugins (p: [ p.aws p.random ]))
    ];
  };
}
```

```
# dns.tf
resource "aws_route53_record" "api" {
  zone_id = "ZXXXXXXXX"
  name    = "api.desk.internal"
  type    = "A"
  ttl     = 60
  records = ["203.0.113.10"]
}
```

The IP is the Colmena `targetHost`. Terraform does not install NixOS (Disko / nixos-anywhere does). Two tools, two state files, one README that says which.

```
nix develop .#tf --command tofu fmt -check
```

Case 5: Promote, then deprecate

```
# channels.nix
{
  channels = {
    desk-26.05 = {
      job = "nixosConfigurations.app-01.toplevel";
      variant = "primary";
      status = "deprecated";
    };
    desk-26.11 = {
      job = "nixosConfigurations.app-01.toplevel";
      variant = "primary";
      status = "stable";
    };
    desk-25.11 = {
      job = "nixosConfigurations.app-01.toplevel";
      variant = "primary";
      status = "unmaintained";
    };
  };
}
```

Unmaintained means: delete the GHA job, leave the table row so `git log` explains why `flake.lock` must not point there. NixOS/infra keeps 25.11 rows as `unmaintained` instead of pretending the files never existed.

```
nix eval --file channels.nix 'channels' --apply builtins.attrNames
```

The trap

The trap is **`cachix push` from every PR** and calling that a channel. A channel is a **named, reviewed** closure (`tested`) plus a status in git. PR pushes belong behind `skipPush`.

The other trap is `checks = mapAttrs every nixosConfigurations` on a laptop flake that also has `aarch64-darwin`. The Linux runner then tries to eval Darwin and fails (or downloads a world). List names.

A third: Terraform `remote-exec` of `nixos-rebuild` so "we only have one tool." That is how state and generations diverge. OpenTofu for DNS/IAM; Colmena for the OS.

The boring rule

- `channels.nix` is a table: job, variant, status. Status changes in git.
- `.tested` (or aggregate) is what CI pushes to the team cache.
- `checks` list host toplevels **by arch**, names explicit.
- OpenTofu `withPlugins` for the cloud envelope. Colmena for NixOS.
- Deprecated deleted. Unmaintained still in CI.

Try this

1. Add `channels.nix` with `desk-26.05.status = "stable"`; `nix eval` it.
2. `nix build .#tested` where `tested` is `app-01`'s toplevel.
3. Write `ci-hosts.nix` with two names; `nix flake check`; add a fake Darwin host name and confirm you did **not** put it on the Linux checks attr.
4. `nix develop .#tf --command which tofu` — store path, plugins attached.
5. In a branch, set `desk-26.05.status = "deprecated"` and read the PR diff as if you were on-call.

Appendices

Glossary of Terms

Glossary of Terms

A concise reference for Nix and NixOS terminology used in this book.

Derivation (`.drv`)

The build specification Nix evaluates to before any compiler runs. An immutable file in `/nix/store` listing the builder, environment, and input paths required to produce an output.

Nix Store (`/nix/store`)

Read-only tree of hashed paths. Identity is the hash; the name-version suffix is a label. Nothing is overwritten in place.

Closure

The store path plus every path it references, transitively. The **runtime** closure is what you copy to production. The **build-time** closure includes compilers and is not what you deploy.

Reference scanning

Nix greps output files for other store hashes. A path mentioned in a binary becomes a runtime reference; gcc used only during the compile does not.

Flake

`flake.nix` + `flake.lock`: hermetic evaluation, pinned inputs, standard outputs (`packages`, `devShells`, `nixosConfigurations`, `checks`).

DevShell

An isolated environment from `nix develop` / `mkShell` with exactly the tools the project listed.

Binary cache / substituter

HTTP service (`cache.nixos.org`, Cachix, Attic, Harmonia) that serves signed NARs. Nix queries substituters before building.

NAR

Nix ARchive: the serialisation of a store path used for copy and substitute.

GC root

A pointer outside the store (`/run/current-system`, `~/.nix-profile`, `result`, `.direnv/flake-profile`) that keeps a closure alive. Unreachable paths are garbage.

Home Manager

NixOS module system aimed at `$HOME`: user packages, dotfiles, user services.

Overlay

Function `final: prev: { … }` that patches `nixpkgs` without editing upstream.

Module system

Typed `options` + merging `config`. Powers NixOS and Home Manager.

Fixed-output derivation (FOD)

A derivation allowed to use the network whose **output hash** you declared. Fetchers (`fetchurl`, `fetchFromGitHub`) are FODs. Identity is the hash, not the URL.

RUNPATH

ELF field pointing at store library directories. Replaces a global `/lib` search.

nix-ld

NixOS compatibility loader at `/lib64/ld-linux-…` so vendor ELFs can find Nix-provided libraries without a global `LD_LIBRARY_PATH`.

FHS env (`buildFHSEnv`)

Bubblewrapped filesystem that looks like `/usr` for vendor installers that refuse to run on a NixOS host.

Remote builder

A machine that realises `.drv` files over SSH (`ssh-ng`). Caches move results; builders move work.

stateVersion

Birth release of a NixOS or Home Manager config. Not an upgrade knob. Leave it until a release note says otherwise.

Generation

One successful `nixos-rebuild switch` or `home-manager switch`. The bootloader (and `home-manager generations`) lists them for rollback.

Impermanence

Root filesystem thrown away at boot; state lives on `/persist`. Requires backups of `/persist`, not of `/nix/store`.

Command Cheat Sheet

Command Cheat Sheet

Daily commands for developing, building, and operating Nix and NixOS.

Flakes and development

Command	Purpose
`nix develop`	Default project shell
`nix develop .#backend`	Named shell
`nix flake init`	Skeleton flake
`nix flake check`	Eval + `checks.*`
`nix flake update`	Refresh every input in `flake.lock`
`nix flake update nixpkgs`	Refresh one input
`nix flake show`	Outputs in this flake
`nix flake metadata`	Locked revisions
`nix fmt`	Formatter if `formatter` is set

Build, run, eval

Command	Purpose
`nix build .#pkg`	Build; symlink `./result`
`nix run .#app`	Build and run
`nix eval .#attr`	Print a value
`nix log ./result`	Build log
`nix repl .#`	REPL on this flake

Inspect closures

Command	Purpose
`nix path-info -Shr ./result`	Closure size (the number that matters)
`nix path-info ./result`	Output path only
`nix-store -q --references P`	Direct runtime edges
`nix-store -q --requisites P`	Full runtime closure
`nix-store -q --referrers P`	Who points at `P`

Command	Purpose
`nix-store -q --tree P`	ASCII graph
`nix why-depends A B`	Why `A`'s closure contains `B`
`nix-tree P`	Interactive tree (`nix run nixpkgs#nix-tree`)
`nom build .#pkg`	`nix build` with a progress table

Fetchers

Command	Purpose
`nix-prefetch-url URL`	Hash a file (packed)
`nix-prefetch-url --unpack URL`	Hash an unpacked archive
`nix hash convert --to sri HASH`	Hex → SRI `sha256-…`

Copy and caches

Command	Purpose
`nix copy --to file://$PWD/nar P`	Directory cache
`nix copy --to ssh://host P`	Copy closure over SSH
`cachix push desk-cache ./result`	Upload to Cachix
`nix show-config | grep substituters`	Which caches this daemon queries

Remote builders

Command	Purpose
`nix build .#pkg --max-jobs 0 --builders 'ssh-ng://user@host system'`	Realise only remotely
`sudo ssh -i /root/.ssh/id_builder user@host nix-store --version`	Daemon SSH must work without a password

NixOS

Command	Purpose
`sudo nixos-rebuild switch --flake .#host`	Build, activate, set boot default
`sudo nixos-rebuild test --flake .#host`	Activate without changing the boot default
`sudo nixos-rebuild boot --flake .#host`	Next reboot only

Command	Purpose
`sudo nixos-rebuild dry-activate --flake .#host`	Show unit changes, do nothing
`sudo nixos-rebuild switch --rollback`	Previous generation
`sudo nix-env --list-generations --profile /nix/var/nix/profiles/system`	Generation list
`nixos-version`	Running release

Home Manager

Command	Purpose
`home-manager switch --flake .#user`	Standalone user env
`home-manager generations`	User generations
`home-manager switch --rollback`	Previous user generation (standalone only)

GC and disk

Command	Purpose
`nix-store --gc --print-roots`	What is keeping paths alive
`nix-collect-garbage`	Delete unreferenced paths; keep generations
`nix-collect-garbage --delete-older-than 14d`	Drop old generations, then orphans
`sudo nix-collect-garbage -d`	All old generations (incident)
`nix-store --optimise`	Hard-link identical files
`nix-store --verify --check-contents`	Rehash after disk trouble
`df -h /nix /boot`	Store vs ESP — two different full-disk incidents

Backup (restic)

Command	Purpose
`sudo systemctl start restic-backups-desk`	Run the NixOS restic unit now
`restic snapshots`	List snapshots (needs `-r` and password file)
`restic restore latest --target /tmp/drill`	Restore drill

Nix and NixOS Version Reference

Nix and NixOS Version Reference

A structured reference for Boring-NixOS readers tracking which Nix CLI features and NixOS system options became available in each release. Coverage spans **Nix 2.18 – 2.35** and **NixOS 23.05 – 26.05**. This book's examples target **Nix 2.35** and **NixOS 26.05 "Yarara"**. Earlier rows are history so you can read old flakes.

Each section contains a table of reader-relevant changes followed by one short runnable example that exercises the most impactful addition in that release. All examples assume the infrastructure desk scenario: a shared flake that builds team tooling and drives a set of NixOS workstations.

Nix CLI Releases

Nix 2.18

Area	Change
`nix flake metadata`	Prints flake description, inputs, and their resolved URLs; last-modified timestamp now shown per input.
`--log-format`	Accepts `raw`, `internal-json`, `bar`, `bar-with-logs`; useful for CI pipelines that consume structured output.
Evaluator	`builtins.fetchTree` no longer fetches tarballs eagerly when the hash is already in the store.
Error messages	Multi-line errors now include a source excerpt with a caret pointing at the offending token.

Key feature — `nix flake metadata`

Save as `show-meta.sh`:

```
# show-meta.sh
nix flake metadata \
  --log-format bar \
  github:NixOS/nixpkgs/nixos-26.05
```

Run:

```
bash show-meta.sh
```

Output:

```
Resolved URL:  github:NixOS/nixpkgs/nixos-26.05
Locked URL:    github:NixOS/nixpkgs/a1b2c3d4e5f6...
Description:   Nix Packages collection & NixOS
Path:          /nix/store/xxxxxxxx-source
Last modified: 2024-11-30 12:00:00
Inputs:
  nixpkgs: (this flake)
```

Nix 2.19

Area	Change
`nix store gc`	Replaces `nix-collect-garbage` progressively; supports `--max-freed <bytes>` to cap space reclaimed per run.
`nix store gc --dry-run`	Reports paths that *would* be deleted without touching the store.
`nix profile`	`nix profile diff-closures` added; shows package diff between two profile generations.
Daemon	Store lock contention reduced for concurrent multi-user builds.

Key feature — `nix store gc --dry-run`

Save as `gc-preview.sh`:

```
# gc-preview.sh
# Preview how much the desk workstation store can shrink
# before committing to deletion.
nix store gc --dry-run --max-freed $((10 * 1024 * 1024 * 1024))
```

Run:

```
bash gc-preview.sh
```

Output:

```
Would delete: /nix/store/aaa...-old-toolchain-2023
Would delete: /nix/store/bbb...-devshell-2023-11
...
Would free 7.4 GiB (limit was 10.0 GiB)
```

Nix 2.20

Area	Change
`nix flake archive`	Copies all flake inputs into a binary cache or a local store, enabling fully offline evaluation afterwards.
`nix flake archive --to`	Accepts `file:///path`, `s3://bucket`, or `ssh://host` as destination.
`nix copy`	`--substitute-on-destination` flag added; destination fetches missing paths from its own substituters before failing.
Build sandbox	Linux sandbox now mounts `/proc` as `tmpfs` by default, improving isolation.

Key feature — `nix flake archive` to a local mirror

Save as `archive-inputs.sh`:

```
# archive-inputs.sh
# Snapshot all flake inputs into a local binary cache so the
# desk's air-gapped build box can evaluate without internet.
nix flake archive \
  --to file:///srv/nix-cache \
  /home/ops/desk-infra
```

Run:

```
bash archive-inputs.sh
```

Output:

```
archiving input 'nixpkgs'
  /nix/store/xxxxxxxx-source → file:///srv/nix-cache
archiving input 'home-manager'
  /nix/store/yyyyyyyy-source → file:///srv/nix-cache
done - 2 inputs archived
```

Nix 2.21

Area	Change
`nix eval --apply`	Applies a Nix function to the evaluated result before printing; enables one-liner transformations.
`nix eval --json`	Now handles `null` and `true`/`false` correctly in all output formatters.
`nix flake update`	`--commit-lock-file` flag commits the updated `flake.lock` with a machine-generated commit message.
Repl	`:reload` re-reads the current file without restarting the session.

Key feature — `nix eval --apply`

Save as `list-packages.sh`:

```
# list-packages.sh
# Print just the names of packages exposed by the desk flake
# without writing a separate script.
nix eval \
  /home/ops/desk-infra#packages.x86_64-linux \
  --apply 'pkgs: builtins.attrNames pkgs' \
  --json
```

Run:

```
bash list-packages.sh
```

Output:

```
["desk-tools","lint-runner","release-packager"]
```

Nix 2.22

Area	Change
`nix build --rebuild`	Rebuilds a derivation even if its output is in the store, then compares the new result to the cached one — a fast reproducibility check.
`nix copy --from` / `--to`	Both flags marked stable; no longer experimental.
`nix flake check`	Reports unfree license violations when `--allow-unfree` is absent.

Area	Change
Evaluator	`builtins.readFileType` added: returns `"regular"`, `"directory"`, `"symlink"`, or `"unknown"`.

Key feature — `nix build --rebuild`

Save as `verify-repro.sh`:

```
# verify-repro.sh
# Rebuild the desk tool and confirm the output hash is stable.
# A mismatch here means a non-deterministic build is lurking.
nix build \
  /home/ops/desk-infra#packages.x86_64-linux.desk-tools \
  --rebuild
```

Run:

```
bash verify-repro.sh
```

Output (reproducible):

```
warning: performing a check build
[3/0/1 built] building desk-tools-1.4.0...
/nix/store/abc123...-desk-tools-1.4.0 (deterministic )
```

Output (non-reproducible):

```
warning: performing a check build
error: derivation '/nix/store/...-desk-tools-1.4.0.drv' may not
       be deterministic: output '/nix/store/abc123...' differs
       from '/nix/store/def456...'
```

Nix 2.23

Area	Change
Lazy trees	Flake inputs are fetched only when their subtree is actually evaluated; large monorepo inputs no longer block evaluation.
`nix fmt`	Declared stable; runs the formatter specified in `flake.nix` `formatter` output.

Area	Change
`nix repl :doc`	Prints the docstring attached to any Nix function or built-in.
`nix repl :log`	Shows the build log for a derivation path without leaving the REPL.
`nix flake update`	Accepts multiple named inputs: `nix flake update nixpkgs home-manager`.

Key feature — `nix fmt` enforced in CI

Save as `fmt-check.sh`:

```
# fmt-check.sh
# Fail CI if any Nix file in the desk flake needs reformatting.
nix fmt /home/ops/desk-infra -- --check
```

Run:

```
bash fmt-check.sh
```

Output (clean):

```
All Nix files already formatted.
```

Output (needs formatting):

```
Would reformat: /home/ops/desk-infra/modules/workstation.nix
error: 1 file would be reformatted
```

Nix 2.24

Area	Change
`nix path-info --json`	Output is now a structured JSON array with `path`, `narHash`, `narSize`, `references`, `registrationTime`, and `deriver` fields.
`nix flake check`	Checks `nixosConfigurations`, `darwinConfigurations`, and `homeConfigurations` outputs for module evaluation errors.
`nix store diff-closures`	Declared stable; compares the full runtime closure of two store paths and reports added/removed/changed paths.

Area	Change
`nix build --print-build-logs`	Default changed to `true` in interactive terminals.

Key feature — `nix store diff-closures`

Save as `diff-release.sh`:

```
# diff-release.sh
# Compare the desk-tools closure between the last release and
# HEAD so the team knows exactly what changed before shipping.
OLD=$(nix build \
  "/home/ops/desk-infra?ref=v1.3.0#packages.x86_64-linux.desk-tools" \
  --no-link --print-out-paths)

NEW=$(nix build \
  /home/ops/desk-infra#packages.x86_64-linux.desk-tools \
  --no-link --print-out-paths)

nix store diff-closures "$OLD" "$NEW"
```

Run:

```
bash diff-release.sh
```

Output:

```
desk-tools: 1.3.0 → 1.4.0
openssl: 3.2.1 → 3.2.2 (+12 KiB)
python3: 3.11.8 → 3.12.3 (+1.1 MiB)
curl: (unchanged)
added: zlib 1.3.1
```

NixOS Releases

NixOS 23.05 — Stoat

Area	Change
systemd	253; `systemd-oomd` enabled by default on desktop profiles.
Python	3.11 is the default interpreter; `python3` symlink resolves to `python3.11`.

Area	Change
Nix	Ships Nix 2.13; flake support via `nix.settings.experimental-features` is opt-in stable.
Module system	`config.system.stateVersion` warning added for stale values.
Security	`services.fail2ban` updated to 1.0; new `ignoreIP` list option.

Key feature — enabling flakes on a Stoat workstation

Save as `configuration.nix`:

```
# configuration.nix
{ pkgs, ... }:
{
  # Enable the two features that make flakes and the new nix
  # command available on every desk workstation running 23.05.
  nix.settings.experimental-features = [
    "nix-command"
    "flakes"
  ];

  # Pin the channel so every workstation evaluates identically.
  nix.registry.nixpkgs.flake = import <nixpkgs> { };

  environment.systemPackages = [ pkgs.git ];
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
building the system configuration...
activating the configuration...
setting up /etc...
the following new units were started: nix-daemon.service
```

NixOS 23.11 — Tapir

Area	Change
GNOME	45; Mutter gains explicit sync support for tear-free rendering on variable-refresh monitors.
PHP	8.2 is default; `services.phpfpm` pools now set `pm.max_spare_servers` automatically.
Kernel	6.1 LTS; default for most hardware profiles.
`services.avahi`	`openFirewall` now defaults to `false`; must be set explicitly.
Nix	Ships Nix 2.18.

Key feature — explicit Avahi firewall declaration

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # 23.11 changed the default: openFirewall is now false.
  # Desk printers and mDNS service discovery need this set
  # explicitly or discovery will silently fail.
  services.avahi = {
    enable = true;
    nssmdns4 = true;   # allow getaddrinfo to resolve .local
    openFirewall = true;
  };
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
activating the configuration...
setting up /etc...
reloading the following units: firewall.service
```

NixOS 24.05 — Uakari

Area	Change
systemd	255; `systemd-boot` gains UKI (Unified Kernel Image) signing support.

Area	Change
Python	3.12 is default; `python3.11` still available as explicit attribute.
Nix	Ships Nix 2.22 — `nix build --rebuild` and stable `nix copy` available out of the box.
`system.etc.overlay`	Experimental option; mounts `/etc` as an overlay for impermanence workflows.
`boot.initrd.systemd`	Available as an opt-in Stage-1; **not** the channel default yet (that lands in 26.05).
`services.postgresql`	`enableJIT` now defaults to `true` for 14+.

Key feature — opt-in systemd initrd (pre-26.05)

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # On 24.05 this is opt-in. 26.05 flips the default to true and
  # deprecates the scripted Stage 1. Prefer the systemd path early
  # if the desk host uses LUKS / TPM2 unlock.
  boot.initrd.systemd.enable = true;
  boot.initrd.luks.devices."cryptroot" = {
    device = "/dev/disk/by-uuid/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx";
    allowDiscards = true;
  };

  # Pin Nix to 2.22 features in this release.
  nix.settings.experimental-features = [
    "nix-command"
    "flakes"
  ];
}
```

Run:

```
sudo nixos-rebuild boot --flake /etc/nixos#desk-build
```

Output:

```
building the system configuration...
updating GRUB 2 menu...
Done. The new configuration will be used when the machine is rebooted.
```

NixOS 24.11 — Vicuna

Area	Change
GNOME	47; default font is now Cantarell 11; GNOME Color Manager updated.
Kernel	6.6 LTS; `amdgpu` gains firmware for RDNA 3.5 integrated graphics.
Nix	Ships Nix 2.24 — `nix path-info --json`, `nix store diff-closures` stable.
`services.openssh.settings`	New sub-options: `MaxSessions`, `PermitUserEnvironment`, `AllowGroups`.
`security.apparmor`	`enableCache` option added; profile compile cache speeds boot on profile-heavy machines.
Python	3.12 remains default; 3.13 available as `python313`.

Key feature — locked-down SSH with `services.openssh.settings`

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # 24.11 exposes AllowGroups directly in the settings attrset,
  # eliminating the need for extraConfig string concatenation.
  services.openssh = {
    enable = true;
    settings = {
      PasswordAuthentication = false;
      PermitRootLogin = "no";
      MaxSessions = 4;
      AllowGroups = [ "ops" "developers" ];
    };
  };

  # AppArmor with compile cache - useful on desk machines that
  # load many profiles at boot.
  security.apparmor = {
    enable = true;
    enableCache = true;
  };
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
activating the configuration...
reloading the following units: sshd.service apparmor.service
```

Nix 2.28 – 2.34 (bridge)

Area	Change
Fetchers	Stricter NAR hash mismatches; prefer SRI `hash = "sha256-…"` over legacy `sha256`.
Flakes	Source trees copied into the store more lazily (completed in 2.35).
`nix store`	`diff-closures`, `copy`, `ping` remain the daily ops set.

Pin `nixos-25.11` only to read old machines. Do not start new desk hosts there: it reached EOL on 2026-06-30.

Nix 2.35

Area	Change
Flake sources	Copied to the store lazily; unused flake files are not hashed into evaluation unless needed.
Sandbox	FreeBSD jail sandboxing; Linux sandbox unchanged.
Installer	Current upstream installer ships 2.35.2 (2026-08).
Security	Recursive-nix advisory fix from the 2.35.0 series.

Key feature — current CLI on a 26.05 host

Stock 26.05 may print `nix (Nix) 2.34.x`. For this book's **2.35+** baseline, install the upstream Nix installer on foreign Linux/macOS, or set `nix.package = pkgs.nixVersions.latest;` (or `nix_2_35`) on NixOS. Then:

```
nix --version
nix flake metadata github:NixOS/nixpkgs/nixos-26.05
```

Output (shape, after pinning 2.35):

```
nix (Nix) 2.35.2
Resolved URL:  github:NixOS/nixpkgs/nixos-26.05
```

NixOS 25.05 / 25.11

25.05 and 25.11 are previous stables. 25.11 "Xantusia" is **deprecated** (EOL 2026-06-30). If `system.stateVersion` is `"25.11"`, **leave it** and move the flake input to `nixos-26.05`.

NixOS 26.05 — Yarara

Area	Change
Support	Bugfix/security until 2026-12-31.
Initrd	Stage-1 systemd initrd is the default (not the old scripted initrd).
Toolchain	GCC 15; GNOME 50 on desktop ISOs.
Nix	Channel ships Nix **2.34**; book CLI baseline remains **2.35+** (installer or `nix.package`).
Modules	New options; confirm names with `nixos-option` if a 24.11 snippet fails eval.

Key feature — pin 26.05 in the desk flake

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk workstation on NixOS 26.05";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.desk-workstation = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        {
          networking.hostName = "desk-workstation";
          system.stateVersion = "26.05";
        }
      ];
    };
  };
```

```
}
```

```
nix flake metadata
sudo nixos-rebuild dry-build --flake .#desk-workstation
```

Quick-Reference Matrix

The table below maps each NixOS release to the Nix CLI it ships and highlights the single most operationally significant change per pair.

NixOS Release	Nix Version Shipped	Most Impactful Addition
23.05 Stoat	2.13	Flakes opt-in stable via `experimental-features`
23.11 Tapir	2.18	`nix flake metadata` improvements; `--log-format` for CI
24.05 Uakari	2.22	`nix build --rebuild` reproducibility check; systemd initrd opt-in
24.11 Vicuna	2.24	`nix store diff-closures` stable; `services.openssh.settings.AllowGroups`
25.05	2.28	`nix flake check` / fetch-tree hardening; module cleanups
25.11 Xantusia	2.31	EOL 2026-06-30; last train before Yarara
26.05 Yarara	2.34 (pin 2.35+)	Current baseline; systemd stage-1 initrd default; GCC 15

Nix 2.19–2.34 intermediate releases are available via `pkgs.nixVersions` even when the channel ships another version. Pin only when you need a CLI feature the channel does not have yet:

```
# configuration.nix  (pin Nix version independently of channel)
{ pkgs, ... }:
{
  nix.package = pkgs.nixVersions.latest; # or pkgs.nixVersions.nix_2_35
}
```

Nix and NixOS Version Reference

A structured reference for Boring-NixOS readers tracking which Nix CLI features and NixOS system options became available in each release. Coverage spans **Nix 2.18 – 2.35** and **NixOS 23.05 – 26.05**. This book's examples target **Nix 2.35** and **NixOS 26.05 "Yarara"**. Earlier rows are history so you can read old flakes.

Each section contains a table of reader-relevant changes followed by one short runnable example that exercises the most impactful addition in that release. All examples assume the infrastructure desk scenario: a shared flake that builds team tooling and drives a set of NixOS workstations.

Nix CLI Releases

Nix 2.18

Area	Change
`nix flake metadata`	Prints flake description, inputs, and their resolved URLs; last-modified timestamp now shown per input.
`--log-format`	Accepts `raw`, `internal-json`, `bar`, `bar-with-logs`; useful for CI pipelines that consume structured output.
Evaluator	`builtins.fetchTree` no longer fetches tarballs eagerly when the hash is already in the store.
Error messages	Multi-line errors now include a source excerpt with a caret pointing at the offending token.

Key feature — `nix flake metadata`

Save as `show-meta.sh`:

```
# show-meta.sh
nix flake metadata \
  --log-format bar \
  github:NixOS/nixpkgs/nixos-26.05
```

Run:

```
bash show-meta.sh
```

Output:

```
Resolved URL:  github:NixOS/nixpkgs/nixos-26.05
Locked URL:    github:NixOS/nixpkgs/a1b2c3d4e5f6...
Description:   Nix Packages collection & NixOS
Path:          /nix/store/xxxxxxxx-source
Last modified: 2024-11-30 12:00:00
Inputs:
  nixpkgs: (this flake)
```

Nix 2.19

Area	Change
`nix store gc`	Replaces `nix-collect-garbage` progressively; supports `--max-freed <bytes>` to cap space reclaimed per run.
`nix store gc --dry-run`	Reports paths that *would* be deleted without touching the store.
`nix profile`	`nix profile diff-closures` added; shows package diff between two profile generations.
Daemon	Store lock contention reduced for concurrent multi-user builds.

Key feature — `nix store gc --dry-run`

Save as `gc-preview.sh`:

```
# gc-preview.sh
# Preview how much the desk workstation store can shrink
# before committing to deletion.
nix store gc --dry-run --max-freed $((10 * 1024 * 1024 * 1024))
```

Run:

```
bash gc-preview.sh
```

Output:

```
Would delete: /nix/store/aaa...-old-toolchain-2023
Would delete: /nix/store/bbb...-devshell-2023-11
...
Would free 7.4 GiB (limit was 10.0 GiB)
```

Nix 2.20

Area	Change
`nix flake archive`	Copies all flake inputs into a binary cache or a local store, enabling fully offline evaluation afterwards.
`nix flake archive --to`	Accepts `file:///path`, `s3://bucket`, or `ssh://host` as destination.
`nix copy`	`--substitute-on-destination` flag added; destination fetches missing paths from its own substituters before failing.
Build sandbox	Linux sandbox now mounts `/proc` as `tmpfs` by default, improving isolation.

Key feature — `nix flake archive` to a local mirror

Save as `archive-inputs.sh`:

```
# archive-inputs.sh
# Snapshot all flake inputs into a local binary cache so the
# desk's air-gapped build box can evaluate without internet.
nix flake archive \
  --to file:///srv/nix-cache \
  /home/ops/desk-infra
```

Run:

```
bash archive-inputs.sh
```

Output:

```
archiving input 'nixpkgs'
  /nix/store/xxxxxxxx-source → file:///srv/nix-cache
archiving input 'home-manager'
  /nix/store/yyyyyyyy-source → file:///srv/nix-cache
done - 2 inputs archived
```

Nix 2.21

Area	Change
`nix eval --apply`	Applies a Nix function to the evaluated result before printing; enables one-liner transformations.
`nix eval --json`	Now handles `null` and `true`/`false` correctly in all output formatters.
`nix flake update`	`--commit-lock-file` flag commits the updated `flake.lock` with a machine-generated commit message.
Repl	`:reload` re-reads the current file without restarting the session.

Key feature — `nix eval --apply`

Save as `list-packages.sh`:

```
# list-packages.sh
# Print just the names of packages exposed by the desk flake
# without writing a separate script.
nix eval \
  /home/ops/desk-infra#packages.x86_64-linux \
  --apply 'pkgs: builtins.attrNames pkgs' \
  --json
```

Run:

```
bash list-packages.sh
```

Output:

```
["desk-tools","lint-runner","release-packager"]
```

Nix 2.22

Area	Change
`nix build --rebuild`	Rebuilds a derivation even if its output is in the store, then compares the new result to the cached one — a fast reproducibility check.
`nix copy --from` / `--to`	Both flags marked stable; no longer experimental.
`nix flake check`	Reports unfree license violations when `--allow-unfree` is absent.

Area	Change
Evaluator	`builtins.readFileType` added: returns `"regular"`, `"directory"`, `"symlink"`, or `"unknown"`.

Key feature — `nix build --rebuild`

Save as `verify-repro.sh`:

```
# verify-repro.sh
# Rebuild the desk tool and confirm the output hash is stable.
# A mismatch here means a non-deterministic build is lurking.
nix build \
  /home/ops/desk-infra#packages.x86_64-linux.desk-tools \
  --rebuild
```

Run:

```
bash verify-repro.sh
```

Output (reproducible):

```
warning: performing a check build
[3/0/1 built] building desk-tools-1.4.0...
/nix/store/abc123...-desk-tools-1.4.0 (deterministic )
```

Output (non-reproducible):

```
warning: performing a check build
error: derivation '/nix/store/...-desk-tools-1.4.0.drv' may not
       be deterministic: output '/nix/store/abc123...' differs
       from '/nix/store/def456...'
```

Nix 2.23

Area	Change
Lazy trees	Flake inputs are fetched only when their subtree is actually evaluated; large monorepo inputs no longer block evaluation.
`nix fmt`	Declared stable; runs the formatter specified in `flake.nix` `formatter` output.

Area	Change
`nix repl :doc`	Prints the docstring attached to any Nix function or built-in.
`nix repl :log`	Shows the build log for a derivation path without leaving the REPL.
`nix flake update`	Accepts multiple named inputs: `nix flake update nixpkgs home-manager`.

Key feature — `nix fmt` enforced in CI

Save as `fmt-check.sh`:

```
# fmt-check.sh
# Fail CI if any Nix file in the desk flake needs reformatting.
nix fmt /home/ops/desk-infra -- --check
```

Run:

```
bash fmt-check.sh
```

Output (clean):

```
All Nix files already formatted.
```

Output (needs formatting):

```
Would reformat: /home/ops/desk-infra/modules/workstation.nix
error: 1 file would be reformatted
```

Nix 2.24

Area	Change
`nix path-info --json`	Output is now a structured JSON array with `path`, `narHash`, `narSize`, `references`, `registrationTime`, and `deriver` fields.
`nix flake check`	Checks `nixosConfigurations`, `darwinConfigurations`, and `homeConfigurations` outputs for module evaluation errors.
`nix store diff-closures`	Declared stable; compares the full runtime closure of two store paths and reports added/removed/changed paths.

Area	Change
`nix build --print-build-logs`	Default changed to `true` in interactive terminals.

Key feature — `nix store diff-closures`

Save as `diff-release.sh`:

```
# diff-release.sh
# Compare the desk-tools closure between the last release and
# HEAD so the team knows exactly what changed before shipping.
OLD=$(nix build \
  "/home/ops/desk-infra?ref=v1.3.0#packages.x86_64-linux.desk-tools" \
  --no-link --print-out-paths)

NEW=$(nix build \
  /home/ops/desk-infra#packages.x86_64-linux.desk-tools \
  --no-link --print-out-paths)

nix store diff-closures "$OLD" "$NEW"
```

Run:

```
bash diff-release.sh
```

Output:

```
desk-tools: 1.3.0 → 1.4.0
openssl: 3.2.1 → 3.2.2 (+12 KiB)
python3: 3.11.8 → 3.12.3 (+1.1 MiB)
curl: (unchanged)
added: zlib 1.3.1
```

NixOS Releases

NixOS 23.05 — Stoat

Area	Change
systemd	253; `systemd-oomd` enabled by default on desktop profiles.
Python	3.11 is the default interpreter; `python3` symlink resolves to `python3.11`.
Nix	Ships Nix 2.13; flake support via `nix.settings.experimental-features` is opt-in stable.
Module system	`config.system.stateVersion` warning added for stale values.
Security	`services.fail2ban` updated to 1.0; new `ignoreIP` list option.

Key feature — enabling flakes on a Stoat workstation

Save as `configuration.nix`:

```
# configuration.nix
{ pkgs, ... }:
{
  # Enable the two features that make flakes and the new nix
  # command available on every desk workstation running 23.05.
  nix.settings.experimental-features = [
    "nix-command"
    "flakes"
  ];

  # Pin the channel so every workstation evaluates identically.
  nix.registry.nixpkgs.flake = import <nixpkgs> { };

  environment.systemPackages = [ pkgs.git ];
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
building the system configuration...
activating the configuration...
setting up /etc...
the following new units were started: nix-daemon.service
```

NixOS 23.11 — Tapir

Area	Change
GNOME	45; Mutter gains explicit sync support for tear-free rendering on variable-refresh monitors.
PHP	8.2 is default; `services.phpfpm` pools now set `pm.max_spare_servers` automatically.
Kernel	6.1 LTS; default for most hardware profiles.
`services.avahi`	`openFirewall` now defaults to `false`; must be set explicitly.
Nix	Ships Nix 2.18.

Key feature — explicit Avahi firewall declaration

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # 23.11 changed the default: openFirewall is now false.
  # Desk printers and mDNS service discovery need this set
  # explicitly or discovery will silently fail.
  services.avahi = {
    enable = true;
    nssmdns4 = true;   # allow getaddrinfo to resolve .local
    openFirewall = true;
  };
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
activating the configuration...
setting up /etc...
reloading the following units: firewall.service
```

NixOS 24.05 — Uakari

Area	Change
systemd	255; `systemd-boot` gains UKI (Unified Kernel Image) signing support.
Python	3.12 is default; `python3.11` still available as explicit attribute.
Nix	Ships Nix 2.22 — `nix build --rebuild` and stable `nix copy` available out of the box.
`system.etc.overlay`	Experimental option; mounts `/etc` as an overlay for impermanence workflows.
`boot.initrd.systemd`	Available as an opt-in Stage-1; **not** the channel default yet (that lands in 26.05).
`services.postgresql`	`enableJIT` now defaults to `true` for 14+.

Key feature — opt-in systemd initrd (pre-26.05)

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # On 24.05 this is opt-in. 26.05 flips the default to true and
  # deprecates the scripted Stage 1. Prefer the systemd path early
  # if the desk host uses LUKS / TPM2 unlock.
  boot.initrd.systemd.enable = true;
  boot.initrd.luks.devices."cryptroot" = {
    device = "/dev/disk/by-uuid/xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx";
    allowDiscards = true;
  };

  # Pin Nix to 2.22 features in this release.
  nix.settings.experimental-features = [
    "nix-command"
    "flakes"
  ];
}
```

Run:

```
sudo nixos-rebuild boot --flake /etc/nixos#desk-build
```

Output:

```
building the system configuration...
updating GRUB 2 menu...
Done. The new configuration will be used when the machine is rebooted.
```

NixOS 24.11 — Vicuna

Area	Change
GNOME	47; default font is now Cantarell 11; GNOME Color Manager updated.
Kernel	6.6 LTS; `amdgpu` gains firmware for RDNA 3.5 integrated graphics.
Nix	Ships Nix 2.24 — `nix path-info --json`, `nix store diff-closures` stable.
`services.openssh.settings`	New sub-options: `MaxSessions`, `PermitUserEnvironment`, `AllowGroups`.
`security.apparmor`	`enableCache` option added; profile compile cache speeds boot on profile-heavy machines.
Python	3.12 remains default; 3.13 available as `python313`.

Key feature — locked-down SSH with `services.openssh.settings`

Save as `configuration.nix`:

```
# configuration.nix
{ ... }:
{
  # 24.11 exposes AllowGroups directly in the settings attrset,
  # eliminating the need for extraConfig string concatenation.
  services.openssh = {
    enable = true;
    settings = {
      PasswordAuthentication = false;
      PermitRootLogin = "no";
      MaxSessions = 4;
      AllowGroups = [ "ops" "developers" ];
    };
  };

  # AppArmor with compile cache - useful on desk machines that
  # load many profiles at boot.
  security.apparmor = {
    enable = true;
    enableCache = true;
  };
}
```

Run:

```
sudo nixos-rebuild switch --flake /etc/nixos#workstation
```

Output:

```
activating the configuration...
reloading the following units: sshd.service apparmor.service
```

Nix 2.28 – 2.34 (bridge)

Area	Change
Fetchers	Stricter NAR hash mismatches; prefer SRI `hash = "sha256-…"` over legacy `sha256`.
Flakes	Source trees copied into the store more lazily (completed in 2.35).
`nix store`	`diff-closures`, `copy`, `ping` remain the daily ops set.

Pin `nixos-25.11` only to read old machines. Do not start new desk hosts there: it reached EOL on 2026-06-30.

Nix 2.35

Area	Change
Flake sources	Copied to the store lazily; unused flake files are not hashed into evaluation unless needed.
Sandbox	FreeBSD jail sandboxing; Linux sandbox unchanged.
Installer	Current upstream installer ships 2.35.2 (2026-08).
Security	Recursive-nix advisory fix from the 2.35.0 series.

Key feature — current CLI on a 26.05 host

Stock 26.05 may print `nix (Nix) 2.34.x`. For this book's **2.35+** baseline, install the upstream Nix installer on foreign Linux/macOS, or set `nix.package = pkgs.nixVersions.latest;` (or `nix_2_35`) on NixOS. Then:

```
nix --version
nix flake metadata github:NixOS/nixpkgs/nixos-26.05
```

Output (shape, after pinning 2.35):

```
nix (Nix) 2.35.2
Resolved URL:  github:NixOS/nixpkgs/nixos-26.05
```

NixOS 25.05 / 25.11

25.05 and 25.11 are previous stables. 25.11 "Xantusia" is **deprecated** (EOL 2026-06-30). If `system.stateVersion` is `"25.11"`, **leave it** and move the flake input to `nixos-26.05`.

NixOS 26.05 — Yarara

Area	Change
Support	Bugfix/security until 2026-12-31.
Initrd	Stage-1 systemd initrd is the default (not the old scripted initrd).
Toolchain	GCC 15; GNOME 50 on desktop ISOs.
Nix	Channel ships Nix **2.34**; book CLI baseline remains **2.35+** (installer or `nix.package`).
Modules	New options; confirm names with `nixos-option` if a 24.11 snippet fails eval.

Key feature — pin 26.05 in the desk flake

Save as `flake.nix`:

```nix
# flake.nix
{
  description = "Desk workstation on NixOS 26.05";

  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";

  outputs = { self, nixpkgs }: {
    nixosConfigurations.desk-workstation = nixpkgs.lib.nixosSystem {
      system = "x86_64-linux";
      modules = [
        {
          networking.hostName = "desk-workstation";
          system.stateVersion = "26.05";
        }
      ];
    };
  };
```

```
}
```

```
nix flake metadata
sudo nixos-rebuild dry-build --flake .#desk-workstation
```

Quick-Reference Matrix

The table below maps each NixOS release to the Nix CLI it ships and highlights the single most operationally significant change per pair.

NixOS Release	Nix Version Shipped	Most Impactful Addition
23.05 Stoat	2.13	Flakes opt-in stable via `experimental-features`
23.11 Tapir	2.18	`nix flake metadata` improvements; `--log-format` for CI
24.05 Uakari	2.22	`nix build --rebuild` reproducibility check; systemd initrd opt-in
24.11 Vicuna	2.24	`nix store diff-closures` stable; `services.openssh.settings.AllowGroups`
25.05	2.28	`nix flake check` / fetch-tree hardening; module cleanups
25.11 Xantusia	2.31	EOL 2026-06-30; last train before Yarara
26.05 Yarara	2.34 (pin 2.35+)	Current baseline; systemd stage-1 initrd default; GCC 15

Nix 2.19–2.34 intermediate releases are available via `pkgs.nixVersions` even when the channel ships another version. Pin only when you need a CLI feature the channel does not have yet:

```
# configuration.nix  (pin Nix version independently of channel)
{ pkgs, ... }:
{
  nix.package = pkgs.nixVersions.latest; # or pkgs.nixVersions.nix_2_35
}
```