OpenTofu

K19G

2026-10-31

Table of contents

OpenTofu **8**
What you will be able to do . . . 8
How the parts are organized . . . 8
The running example . . . 9
What this book leaves out . . . 9

First Principles **10**

The OpenTofu Story: From Terraform to Open Source **11**

The OpenTofu Story: From Terraform to Open Source **12**
Learning goals . . . 12
What Is OpenTofu? . . . 12
The Terraform License Change: A Turning Point . . . 12
HashiCorp vs. Linux Foundation: The Fork That Changed Everything . . . 13
OpenTofu vs. Terraform: Compatibility, Differences, and Trade-offs . . . 13
When to Choose OpenTofu (and When Not To) . . . 14
Summary . . . 14
Sources . . . 15

Architecture and Core Concepts **16**

Architecture and Core Concepts **17**
Learning goals . . . 17
The Declarative Paradigm: What You Define vs. What Happens . . . 17
OpenTofu's Execution Model: Plan, Apply, Destroy . . . 17
Providers: The Bridge Between Configuration and Reality . . . 18
Resources: The Fundamental Building Blocks . . . 18
State: Why It Exists and How It Works . . . 19
Data Sources: Reading the World Without Creating It . . . 19
Summary . . . 19
A configuration that is only these ideas . . . 20

Installation and Project Structure **21**

Installation and Project Structure **22**
Learning goals . . . 22
Installing OpenTofu: Linux, macOS, Windows, and Containerized Options . . . 22
Verifying Your Installation and Checking Versions . . . 23
The Minimal Working Project . . . 23

Directory Layout Conventions: Single Project vs. Modular Organization 24
Configuration File Naming Rules: main.tf, variables.tf, and Beyond 24
.terraform-version, .gitignore, and Essential Project Files 24
Summary 25

Language **26**

HCL Syntax and Language Fundamentals **27**

HCL Syntax and Language Fundamentals **28**
Learning goals 28
Blocks, Arguments, Comments, and Basic Structure 28
Attributes and Types: Strings, Numbers, Booleans, Lists, Maps, Objects 28
Expressions: Interpolation, Conditionals, and Operators 29
For Expressions: Transforming Collections Elegantly 30
Dynamic Blocks: Generating Repetitive Configuration Programmatically 30
Null, Unknown, and Type Conversion: Gotchas Every Engineer Must Know 31
Summary 31

Variables, Locals, and Outputs **32**

Variables, Locals, and Outputs **33**
Learning goals 33
Input Variables: Types, Defaults, Validation Rules, and Sensitive Values 33
Variable Files (.tfvars): Environment-Specific Configuration Management 34
Locals: Organizing Complex Logic and Avoiding Repetition 34
Outputs: Exposing Information and Connecting Modules 35
Passing Data Between Modules: The Parent-Child Contract 35
Summary 36

Providers and State **37**

Providers, Resources, and Data Sources in Depth **38**

Providers, Resources, and Data Sources in Depth **39**
Learning goals 39
How Providers Work: Plugins, Authentication, and Configuration Blocks 39
Resource Arguments: Required, Optional, Computed, and Deprecated 40
Creating Resources Across Clouds: AWS, Azure, GCP Examples 40
Data Sources: Querying Existing Infrastructure Safely 41
Provider Aliases: Managing Multiple Configurations Simultaneously 41
Version Constraints: Pinning Providers for Stability 42
Summary 43

State Management and Backends **44**

State Management and Backends **45**
Learning goals 45

Understanding State: What It Stores and Why Local Is Not Enough . . . 45
Backend Types: Local, S3, Azure Blob, GCS, Consul, and More . . . 45
Remote State Configuration: Locking, Encryption, and Versioning . . . 46
Reading Remote State: Cross-Referencing Infrastructure Between Projects . . . 47
State Migration: Moving Between Backends Safely . . . 48
State Troubleshooting: Corruption, Drift, and Recovery Procedures . . . 49
Summary . . . 49

Composition **50**

Modules: Building Reusable Infrastructure Components **51**

Modules: Building Reusable Infrastructure Components **52**
Learning goals . . . 52
What Is a Module?: Root Modules, Child Modules, and Published Modules . . . 52
Designing Your First Module: Inputs, Outputs, and Responsibility Boundaries . . . 52
Calling Modules: Local Paths, Registry Sources, and Git Repositories . . . 53
The OpenTofu Registry: Publishing and Consuming Community Modules . . . 54
Advanced Module Patterns: Stacks, Composition, and Versioned Releases . . . 54
Anti-Patterns: Monolithic Modules, Hidden Dependencies, and Over-Abstraction . . . 55
Summary . . . 55

Refactoring Without Replacement: moved, removed and Deprecations **56**

Refactoring Without Replacement: moved, removed and Deprecations **57**
Learning goals . . . 57
`moved`: change the address, keep the object . . . 57
Changing how many: instance keys . . . 58
Modules . . . 59
Splitting a module . . . 59
What `moved` cannot do . . . 60
The lifetime of a `moved` block . . . 60
`removed`: stop managing, keep the object . . . 60
Handover recipe: moving an object to another configuration . . . 61
`deprecated`: warn callers first . . . 62
Choosing the tool . . . 62
Diagnosis . . . 62
Operational care . . . 63
Practice . . . 63
Recall card . . . 63
Sources . . . 64

Workspaces, Lifecycle Rules, and Dependencies **65**

Workspaces, Lifecycle Rules, and Dependencies **66**
Learning goals . . . 66
Workspaces: Multiple Environments from One Configuration . . . 66
When Workspaces Help (and When They Hurt): Practical Guidance . . . 66

Lifecycle Arguments: create_before_destroy, prevent_destroy, ignore_changes 67
Dependency Management: Explicit depends_on vs. Implicit References 68
The Dependency Graph: Visualizing and Debugging Execution Order 68
Tainting Resources: Forcing Recreation When Needed 68
Summary 69

Provider Instances with for_each and the enabled Meta-Argument 70

Provider Instances with for_each and the enabled Meta-Argument 71
Learning goals 71
Static aliases do not scale 71
Selecting an instance 72
One module instance per region 72
The removal rule 73
`enabled`: zero or one, without `[0]` 74
Migrating from count 75
Choosing the construct 76
Diagnosis 76
Operational care 76
Practice 77
Recall card 77
Sources 77

Provisioners and Importing Existing Infrastructure 78

Provisioners and Importing Existing Infrastructure 79
Learning goals 79
The Problem With Provisioners: Why They Are Discouraged 79
When Provisioners Make Sense: file, local-exec, remote-exec, and Beyond 79
Best Practices If You Must Use Provisioners: Idempotency and Error Handling 80
Importing Existing Resources: The import Command and Workflow 80
Generating Configuration for Imported Resources: State Inspection Techniques 81
Migration Strategies: Bringing Legacy Infrastructure Under OpenTofu Control 81
Summary 82

Confidence 83

Testing, Validation, Formatting, and Policy Enforcement 84

Testing, Validation, Formatting, and Policy Enforcement 85
Learning goals 85
Formatting and Linting: tofu fmt, tofu validate, and Custom Rules 85
Pre-commit Hooks: Catching Errors Before They Reach the Repository 86
Testing Strategies: Unit Tests, Integration Tests, and Plan Validation 86
OpenTofu Test Framework: Built-in Testing Capabilities 86
Policy as Code: Sentinel, OPA, and openpolicyagent Integration 87
Security Scanning: Detecting Misconfigurations with tfsec, checkov, and Trivy 87
Summary 88

Security Best Practices and Secrets Management **89**

Security Best Practices and Secrets Management **90**
Learning goals . . . 90
Securing State Files: Encryption at Rest and in Transit . . . 90
Managing Secrets: Never Hardcoding Credentials . . . 91
Provider Authentication Patterns: IAM Roles, Service Principals, Workload Identity . . . 91
Least Privilege for OpenTofu Execution: IAM Policies and RBAC . . . 92
Audit Logging and Compliance: Tracking Who Changed What and When . . . 92
Common Security Pitfalls: Exposed Keys, Overly Permissive Policies, and Insecure Backends 93
Summary . . . 93

Ephemeral Values and Write-Only Attributes **94**

Ephemeral Values and Write-Only Attributes **95**
Learning goals . . . 95
Three different promises . . . 95
Where an ephemeral value may go . . . 96
Ephemeral input variables . . . 96
Ephemeral resources . . . 96
Write-only attributes . . . 97
Changing a write-only value . . . 97
Ephemeral outputs . . . 98
A complete flow . . . 98
Diagnosis . . . 99
Operational care . . . 99
Practice . . . 100
Recall card . . . 100
Sources . . . 100

Enterprise **101**

CI/CD Integration, GitOps, and Automation Workflows **102**

CI/CD Integration, GitOps, and Automation Workflows **103**
Learning goals . . . 103
The Infrastructure Pipeline: Commit, Plan, Review, Apply . . . 103
GitHub Actions Integration: Complete Workflow Examples . . . 104
GitLab CI, Jenkins, and Azure DevOps: Multi-Platform Pipeline Patterns . . . 105
GitOps With OpenTofu: Declarative Deployment Using ArgoCD and Flux . . . 105
Approval Gates and Change Management: Human Review in Automated Workflows . . . 106
Parallel Execution and Performance Optimization: Scaling Plans and Applies . . . 106
Summary . . . 107

Enterprise-Scale Patterns and Production Operations **108**

Enterprise-Scale Patterns and Production Operations **109**
Learning goals . . . 109

Multi-Cloud and Hybrid Infrastructure: Managing AWS, Azure, GCP, and On-Premises Together . . . 109
Organizational Structure: Team-Based Repository Layouts and Shared Responsibility . . 110
Environment Strategy: Development, Staging, Production Separation Patterns 110
Versioning and Upgrades: OpenTofu Versions, Provider Updates, and Breaking Changes . 111
Debugging and Logging: Tracing Failures in Complex Deployments 111
Real-World Case Studies: Production Patterns from Leading Organizations 112
Summary 112

Conclusion: Delivering on the Infrastructure-as-Code Promise **113**

Conclusion: Delivering on the Infrastructure-as-Code Promise **114**
What you can do this quarter 114
What the promise does not include 114
From first principles to production 115

OpenTofu

Infrastructure as code for modern cloud and on-premises environments. This book starts from what a configuration means and ends at a production change pipeline.

The command is `tofu`. Examples target **OpenTofu 1.13.2** (the 1.13 series, supported until 1 August 2027). The block that declares versions is still named `terraform`, so existing modules keep loading. Worked commands assume Linux. Chapter 3 also gives the macOS, Windows, and container installs.

> **Note**
>
> OpenTofu is a community fork of Terraform, published under the Mozilla Public License 2.0 and hosted by the Linux Foundation. It is not a HashiCorp product. Where the two tools have diverged, this book says so.

What you will be able to do

- Explain why OpenTofu exists, and when a Terraform 1.6+ feature will not load
- Write HCL that plans the same way twice: variables, locals, outputs, and modules
- Configure providers, remote state, locking, and client-side state encryption
- Import an existing resource and test a module with `tofu test`
- Keep credentials out of state reviews, and run plan/apply from a pipeline with a human gate

How the parts are organized

Part	Chapters	You leave able to
01 First principles	1–3	Install 1.13.2 and explain plan, apply, state, and providers
02 Language	4–5	Read and write HCL, variables, locals, and outputs
03 Providers and state	6–7	Pin providers and store state somewhere other than a laptop
04 Composition	8–10	Build modules, choose workspaces honestly, and import what already exists
05 Confidence	11–12	Format, test, scan, and stop secrets landing in git
06 Enterprise	13–14 and the close	Run a pipeline and a multi-environment layout

The running example

Chapters that need a real apply use the `hashicorp/random` provider and a `random_pet` resource. That needs no cloud account. AWS, Azure, and GCP snippets are configuration sketches: they show the arguments, and they are not a substitute for a credentials setup you have tested.

What this book leaves out

- A certification workbook
- Click-paths in a vendor console
- HashiCorp's commercial Terraform Cloud / HCP Terraform product surface. Sentinel, in particular, does not run on OpenTofu. Chapter 11 uses Open Policy Agent instead

First Principles

The OpenTofu Story: From Terraform to Open Source

The OpenTofu Story: From Terraform to Open Source

OpenTofu is the open-source continuation of the Terraform language and workflow, under the Mozilla Public License 2.0. This chapter is the history you need before you trust a module written for the other binary.

Learning goals

- Say what OpenTofu is, and what it deliberately kept from Terraform
- Place the August 2023 license change on a timeline
- Tell a 1.5-era configuration from a feature that exists on only one of the two tools
- Choose OpenTofu for a reason you can write down

What Is OpenTofu?

OpenTofu is a command-line tool, `tofu`, that reads HashiCorp Configuration Language (HCL) and reconciles it with infrastructure through provider plugins. You declare resources. It builds a plan. You apply the plan. It records what it created in a state file.

The project is a fork of Terraform as it stood under MPL 2.0. The Linux Foundation hosts it. In April 2025 the project was accepted into the CNCF sandbox. Providers are not rewritten by OpenTofu. The same provider plugins Terraform uses are published for the OpenTofu registry at `registry.opentofu.org`, and the protocol between `tofu` and those plugins is the Terraform plugin protocol.

This book pins **1.13.2**, released 21 October 2026. The 1.13 series is the supported line until 1 August 2027. Terraform's version numbers are not OpenTofu's. In the same month Terraform's current release was in the 1.17 series. Equal numbers do not mean equal features, and a higher Terraform number does not mean OpenTofu is behind on every axis. State encryption shipped in OpenTofu 1.7 and is not a Terraform language feature.

The Terraform License Change: A Turning Point

Through Terraform 1.5.x, HashiCorp shipped Terraform under MPL 2.0. On 10 August 2023 HashiCorp moved future releases of Terraform, and of several other products, to the Business Source License (BSL, also written BUSL) 1.1. The BSL is source-available. It is not an Open Source Initiative license. It restricts using the licensed software to offer a competing product.

1.5.x remained the last MPL release. Configurations, state, and the provider protocol up to that line are the common ancestor. Everything HashiCorp shipped as Terraform 1.6 and later is under the BSL. Everything OpenTofu shipped is under MPL 2.0, starting from that ancestor and then diverging on purpose.

The practical effect for an engineer: you can read either codebase, and your lawyer will care which binary you ship inside a product. The practical effect for a module: a file that only uses language from 1.5 loads in both. A file that uses a feature added on one side after the fork needs the binary that has it.

HashiCorp vs. Linux Foundation: The Fork That Changed Everything

The fork was announced as an open-source Terraform, then named OpenTofu, and contributed to the Linux Foundation so the trademark and the governance were not a single vendor's. The first stable OpenTofu release was 1.6 in January 2024, compatible with the 1.5 language and ready to keep moving.

Governance is the part that is easy to skip and expensive to misunderstand. HashiCorp (later under IBM) decides Terraform's product direction, including HCP Terraform, Stacks, and Sentinel. The OpenTofu technical steering committee decides OpenTofu's direction in public issues and RFCs. A feature that is a commercial control plane on one side may be a CLI feature, or absent, on the other.

The registry split follows the same line. `registry.terraform.io` is HashiCorp's. `registry.opentofu.org` is the community registry. Provider namespaces such as `hashicorp/aws` exist on both because the providers themselves stayed available. When you write `source = "hashicorp/aws"` in this book, `tofu init` resolves it through the OpenTofu registry unless you override the host.

OpenTofu vs. Terraform: Compatibility, Differences, and Trade-offs

Treat compatibility as three layers.

Layer	What still matches	What has diverged
Language shared with 1.5	Resources, modules, `count`, `for_each`, import blocks, checks, the testing framework's early form	—
State	State written by Terraform through 1.5.x is the documented drop-in. Later state can often be migrated, and you test that migration. You do not assume it	Client-side encryption of state and plan files is OpenTofu (1.7+)

Layer	What still matches	What has diverged
Later language	Both grew ephemeral resources, write-only attributes, and `provider for_each` on their own timelines	OpenTofu 1.13 adds `convert` and `assumenotnull` to tame unknown values at plan time. Terraform 1.13+ added Stacks, a query workflow, and a commercial policy product. Version numbers crossed and the features did not

Other OpenTofu marks you will meet in later chapters: the `.tofu` file extension (1.8) for files that must not be read by Terraform, variables and locals inside module sources and backends (1.8), OCI registry addresses (1.10), S3 locking without a separate lock table (1.10), and the `enabled` meta-argument (1.11).

1.13 also removes WinRM from the `file` and `remote-exec` provisioners, changes `base64gzip` output bytes, requires macOS 13 or newer, and drops official 32-bit builds. Windows on ARM64 is a supported package. Read the series changelog before you bump a production pin. Chapter 14 is that upgrade habit.

When to Choose OpenTofu (and When Not To)

Choose OpenTofu when the license has to be MPL, when you want client-side state encryption in the CLI, or when you want the roadmap set by a foundation project rather than a commercial control plane. Choose it when your modules stay within the shared language, or within OpenTofu features you have pinned.

Stay on Terraform when a system you already pay for requires the Terraform binary: an HCP Terraform workspace, a Sentinel policy pack you are not ready to rewrite as Open Policy Agent, or a module that uses a Terraform-only feature you have confirmed OpenTofu rejects. "We might use Stacks later" is not a reason. A failing `tofu init` on a required feature is.

Do not run both binaries against the same state. Pick one per state file, pin it in `.terraform-version` or your runtime manager, and review the plan under that pin.

Summary

- OpenTofu is the MPL fork of Terraform's 1.5-era CLI, hosted by the Linux Foundation
- The BSL change on 10 August 2023 is why the fork exists
- Shared language is not shared version numbers. This book pins 1.13.2
- State encryption is an OpenTofu feature. Sentinel and HCP Terraform are not
- One state file, one binary

Sources

- OpenTofu project: https://opentofu.org/
- Mozilla Public License 2.0: https://www.mozilla.org/MPL/2.0/
- OpenTofu releases and the 1.13 changelog: https://github.com/opentofu/opentofu/releases

Architecture and Core Concepts

Architecture and Core Concepts

Six words carry the rest of the book: configuration, plan, state, provider, resource, data source. This chapter defines them before any cloud account appears.

Learning goals

- Separate what you write from what `tofu` does
- Walk init, plan, apply, and destroy without memorising flags
- Explain why state exists
- Tell a resource from a data source

The Declarative Paradigm: What You Define vs. What Happens

You write the desired end state. You do not write the API calls. A resource block says "this object should exist with these arguments." OpenTofu's job is to notice the difference between that text and the recorded reality, and to call the provider until the difference is gone or the plan says it cannot be done without a destructive change.

That is declarative in the useful sense, not in the absolute sense. The order of blocks in a file is not the order of API calls. A `local-exec` provisioner is an imperative escape hatch, which is why chapter 10 discourages it. Arguments you set are desired state. Attributes the provider fills in (an id, a generated name) are observed state. Mixing those up is the first source of plans that fight you.

Idempotence is the promise: apply, then apply again, and the second plan is empty when nothing outside the configuration changed. Drift is the world editing itself. A person in a console, or another tool, changes an object the state still tracks. The next plan proposes to put it back. That proposal is the point of the tool.

OpenTofu's Execution Model: Plan, Apply, Destroy

The commands you will actually live in:

Command	Reads	Writes	Talks to the infrastructure
`tofu init`	`required_providers`, backend, modules	`.terraform/`, lock file	Downloads providers and modules

Command	Reads	Writes	Talks to the infrastructure
`tofu plan`	Configuration, state, data sources	A plan file if you ask	Refreshes and computes a diff
`tofu apply`	A plan, or it makes one and waits	State	Creates, updates, and destroys objects
`tofu destroy`	Configuration and state	State	Destroys what state tracks

`plan` is a dry run with teeth. It refreshes state by default, so it does contact APIs. It does not change infrastructure. `apply` with no plan file plans and then prompts. In automation you save the plan (`tofu plan -out=tfplan`) and apply that exact file (`tofu apply tfplan`) so the change a human approved is the change that runs.

Destroy is not the opposite of a git revert. It deletes the objects in state. A revert of the configuration, followed by apply, also deletes objects that disappeared from the configuration. Both are destructive. Chapter 9's `prevent_destroy` is how you make a specific resource refuse.

Providers: The Bridge Between Configuration and Reality

OpenTofu does not know what an S3 bucket is. A provider plugin does. `tofu` starts the plugin, hands it the resource configuration, and stores whatever identity the plugin returns.

The `required_providers` block names the source and the version. Init downloads the plugin and writes `.terraform.lock.hcl`. That lock file is how the next machine gets the same plugin build. Commit it. Chapter 6 is authentication, aliases, and version constraints. For now, the bridge is: configuration in, API calls out, attributes back.

A provider configuration block (`provider "aws" { ... }`) is not a resource. It is credentials, region, and endpoints for every resource of that type. No block is required for providers that need no credentials. `random` is the one this book applies for real.

Resources: The Fundamental Building Blocks

A resource has a type and a local name: `resource "random_pet" "service"`. The address you use everywhere else is `random_pet.service`. The type is the provider's. The local name is yours, unique among that type in the module.

Arguments are what you set. After apply, the state holds those plus computed attributes. The next plan compares both. Changing an argument that the provider cannot update in place produces a destroy and a create. The plan says so with `-/+` or a replacement. Read that line before you approve it.

`count` and `for_each` make many instances from one block. Instances have addresses like `random_pet.service["api"]`. Those addresses are how state finds the object. Renaming the key is a destroy and a create unless you move the state. Chapter 9 covers `moved` blocks and dependencies.

State: Why It Exists and How It Works

The configuration says what you want. It does not remember the id of the object it created last Tuesday. State does. State is a JSON document mapping each resource address to a provider-specific id and the attribute values from the last refresh.

Without state, the next apply would try to create a second copy, or would have to guess which remote object belongs to which block. Guessing is how tools delete the wrong thing. State is the memory that makes the guess unnecessary.

Local state is a file named `terraform.tfstate` in the working directory. It is enough for the pet example. It is the wrong place for anything a second person, or a pipeline, must update. Two applies against two copies of state create two worlds. Chapter 7 moves state to a backend with locking. State also stores attribute values, including ones marked sensitive. Sensitive means "do not print it in the plan UI." It does not mean "omit it from the state file."

Data Sources: Reading the World Without Creating It

A data source reads. It does not create, update, or destroy. `data "aws_ami" "latest"` asks the provider a question and stores the answer for the rest of the configuration to interpolate.

Data sources are refreshed during plan. A data source that changes under you changes the plan, which is what you want when you are looking up a current AMI, and what you do not want when the lookup is a moving target you should have pinned. Prefer a resource for anything you created and now need to remember. Prefer a data source for something that already existed and is owned elsewhere.

Data sources land in state too, as a cache of the last read. They are not a second source of truth. The next refresh overwrites them.

Summary

- You declare the end state. OpenTofu computes the API calls
- Init fetches plugins. Plan diffs. Apply changes. Destroy deletes what state tracks
- Providers speak APIs. Resources are the objects. State remembers their ids
- A data source reads. It does not own the object
- A second apply is empty only when the world still matches the configuration and the state

A configuration that is only these ideas

```
terraform {
  required_version = "~> 1.13.0"

  required_providers {
    random = {
      source  = "hashicorp/random"
      version = "~> 3.6"
    }
  }
}

resource "random_pet" "service" {
  length = 2
}

output "name" {
  value = random_pet.service.id
}
```

`tofu init`, `tofu plan`, and `tofu apply` against that file are the whole execution model with no account attached.

Installation and Project Structure

Installation and Project Structure

Pin the binary before you pin the providers. This chapter installs OpenTofu 1.13.2, checks that the binary is that release, and lays out a repository that a second engineer can clone.

Learning goals

- Install 1.13.2 on Linux, and know the macOS, Windows, and container equivalents
- Confirm the version before trusting a plan
- Create a one-resource project that applies
- Name files so a stranger can find variables, and ignore the files that must not be committed

Installing OpenTofu: Linux, macOS, Windows, and Containerized Options

The worked path in this book is Linux, with the version selected by a runtime manager so the repository and the laptop agree.

Linux, pinned with mise. mise ships an OpenTofu backend. From the project directory:

```
mise use opentofu@1.13.2
mise exec -- tofu version
```

A global install is `mise use -g opentofu@1.13.2`. The project file mise writes belongs in git. A laptop without mise can install the same release from the official installer, which also configures the project's apt or rpm repository:

```
curl --proto '=https' --tlsv1.2 -fsSL \
  https://get.opentofu.org/install-opentofu.sh -o install-opentofu.sh
chmod +x install-opentofu.sh
./install-opentofu.sh --install-method deb
```

`--install-method rpm` is the dnf/yum equivalent. The package name is `tofu`. An apt repository that tracks "latest" will walk off 1.13 the day 1.14 ships. A pin beats a floating distro package for this book.

macOS. Homebrew's `brew install opentofu` is the OpenTofu tap and tracks current. For 1.13.2 specifically, use mise or the release tarball. 1.13 requires macOS 13 or newer. Official 32-bit builds are gone.

Windows. Install the signed 1.13.2 package from the OpenTofu GitHub release, or the OpenTofu package in winget, and confirm the version in the next section. Windows on ARM64 is supported in 1.13. There is no official 32-bit build. WinRM provisioners are gone in this series. Use OpenSSH if a provisioner must reach Windows, and prefer not to use a provisioner at all (chapter 10).

Containers. The project publishes images at `ghcr.io/opentofu/opentofu`. Pin the tag:

```
docker run --rm ghcr.io/opentofu/opentofu:1.13.2 version
```

Mount the working directory when you init or plan. `latest` is not a pin. The `minimal` tag is the one to copy a binary out of when you build your own image.

Verifying Your Installation and Checking Versions

`tofu version` prints the release and the platform. You want `1.13.2` on the platform you will apply from. A plan created on 1.13.2 should be applied by 1.13.2. Providers have their own versions, printed by `tofu version` after init and locked in `.terraform.lock.hcl`.

Three versions are in play, and mixing them up wastes an afternoon:

Version	Where it lives	What breaks if it floats
OpenTofu	`tofu version`, `.terraform-version`, mise	Language and state behaviour
Provider	`required_providers` and the lock file	Arguments the resource accepts
Module	`source` and `version` on a `module` block	Inputs the child expects

`required_version = "~> 1.13.0"` refuses to run on 1.12 and on 1.14. That pessimism is correct for a book pin. Widen it only when you have read the changelog.

The Minimal Working Project

Create a directory with one file, `main.tf`, containing the `random_pet` configuration from chapter 2. Then:

```
tofu init
tofu plan -out=tfplan
tofu apply tfplan
tofu output name
```

Init creates `.terraform/` and `.terraform.lock.hcl`. Plan writes a binary plan. Apply updates `terraform.tfstate`. Output prints the pet name. A second `tofu plan` should report no changes.

Delete the experiment with `tofu destroy` when you are done with it. The state file is not a secret worth keeping for a pet name, and it is a secret-shaped file the moment a real resource lands in it.

Directory Layout Conventions: Single Project vs. Modular Organization

A single project is one root module: the directory where you run `tofu`.

```
network/
   main.tf            # resources, or a thin file that calls modules
   variables.tf       # input variables
   outputs.tf         # outputs
   versions.tf        # required_version and required_providers
   providers.tf       # provider configuration
   terraform.tfvars   # values for this environment, if not secret
   .terraform.lock.hcl
   .terraform-version
```

Filenames are convention. OpenTofu loads every `.tf` and `.tofu` file in the directory and ignores the names. The split exists so a reviewer knows where to look.

A modular organisation pulls reusable blocks into child modules (chapter 8) and keeps each environment or each account as its own root module with its own state. That is the layout chapter 14 uses for production. Start in one directory until the pet example is boring. Splitting early hides the tool behind a forest of folders.

Configuration File Naming Rules: main.tf, variables.tf, and Beyond

File	Put it here
`versions.tf`	`terraform` block: version, providers, backend, encryption
`variables.tf`	`variable` blocks only
`outputs.tf`	`output` blocks only
`providers.tf`	`provider` blocks
`main.tf`	Resources, or module calls, when the file stays readable
`*.tfvars`	Values. Not logic
`*.tofu`	OpenTofu-only syntax you must not let Terraform parse

Do not use override files (`override.tf`) to smuggle a local tweak. They merge in a way reviewers miss. A variable does the same job in the open.

JSON (`.tf.json`) is valid and a poor authoring format. Generate HCL, or generate JSON from a tool and review the diff. Hand-written JSON configurations rot.

.terraform-version, .gitignore, and Essential Project Files

`.terraform-version` contains `1.13.2`. tfenv, tofuenv, and tenv read it. mise users can rely on `mise.toml` instead. One pin, not both, unless you have checked they agree.

Commit:

- `*.tf`, `*.tofu`, `*.tfvars` that hold no secrets
- `.terraform.lock.hcl`
- `.terraform-version` or `mise.toml`
- A `README` with the init/plan/apply commands

Never commit:

- `.terraform/` (provider binaries)
- `terraform.tfstate` and `terraform.tfstate.backup`
- `*.tfplan` and `tfplan` (plans can contain sensitive values)
- `crash.log`
- `*.auto.tfvars` or `secret.auto.tfvars` if they hold credentials
- `.terraformrc` when it contains tokens

A `.gitignore` that encodes that list belongs in the first commit, before the first apply. The expensive mistake is committing a state file once. History keeps it after you delete the working copy.

`TF_VAR_name` environment variables fill variable `name`. Prefer them for secrets. Chapter 12 is the full rule. Chapter 5 is how variables are declared so that path works.

Summary

- Pin 1.13.2 with mise, the release image, or an installer, then run `tofu version`
- `~> 1.13.0` keeps the series and refuses the next one
- One root module is enough until the example applies cleanly twice
- Names of `.tf` files are a human convention. The lock file and the version pin are not
- State, plans, and `.terraform/` stay out of git

Language

HCL Syntax and Language Fundamentals

HCL Syntax and Language Fundamentals

HCL is a small language with a few traps that show up as a plan that cannot be computed. This chapter is the syntax you need to read anyone else's module.

Learning goals

- Parse a block into type, labels, and arguments
- Choose a type instead of stringifying everything
- Write a `for` expression and a `dynamic` block when repetition is data
- Predict what `null` and an unknown value will do to `count`

Blocks, Arguments, Comments, and Basic Structure

A block has a type, zero or more labels, and a body.

```
resource "random_pet" "service" {
  length    = 2
  separator = "-"
}
```

`resource` is the type. `"random_pet"` and `"service"` are labels. `length` and `separator` are arguments. Argument order does not matter. A missing required argument is an error at plan time. An unknown argument is also an error, which is how typos die early.

`#` and `//` comment to end of line. `/* */` comments a range. Comments are not documentation the registry renders. Prefer a `description` on variables and outputs for anything a caller must see.

Identifiers are letters, digits, underscores, and hyphens in some positions. Resource local names should be stable. Renaming `service` to `api` is a new address, and state will want to destroy the old one.

Attributes and Types: Strings, Numbers, Booleans, Lists, Maps, Objects

Type	Literal	Use
string	`"api"`	Names, ids you were given
number	`2`	Counts and sizes. Do not quote numbers you intend to do arithmetic on
bool	`true`	Flags. The strings `"true"` and `"false"` are strings
list	`["a", "b"]`	Ordered, one element type in simple declarations
set	`toset(["a", "a"])`	Unordered, unique. A set of strings is the natural `for_each`
map	`{ env = "prod" }`	String keys, one value type
object	`{ name = "api", port = 443 }`	Named attributes with their own types
tuple	`["api", 443]`	Fixed positions, mixed types

A list and a tuple both use `[]`. The type constraint you declare is what makes them different. Chapter 5's variable types are these same types with validation on top.

Heredocs (`<<-EOT` … `EOT`) are strings. The closing marker sits at the left margin unless you use the dashed form, which strips the indentation of the least-indented line.

Expressions: Interpolation, Conditionals, and Operators

Inside strings, `${...}` interpolates. `"pet-${random_pet.service.id}"` is a string. Outside strings, you do not wrap references in quotes or in `${}`. `random_pet.service.id` is the value. `"random_pet.service.id"` is the text of that address and a classic bug.

Conditionals are `condition ? then_value : else_value`. Both branches are typed together. A branch that returns a string and a branch that returns a number will fail or coerce in a way you will not like. Make them the same type.

`&&`, `||`, and `!` are boolean. `+`, `-`, `*`, `/` are numeric. `==` and `!=` compare. Comparisons against `null` are how optional values show up. Operator precedence will surprise you less if you parenthesise the conditional.

A splat `aws_instance.web[*].id` turns a list of resources into a list of attributes. On a resource that uses `for_each`, the value is a map, and you want a `for` expression rather than a splat.

For Expressions: Transforming Collections Elegantly

A `for` expression builds a new collection. It does not loop at apply time. It is evaluated while the configuration is being built.

```
locals {
  ports = [80, 443]

  listeners = [
    for port in local.ports : {
      port     = port
      protocol = port == 443 ? "https" : "http"
    }
  ]

  by_protocol = {
    for listener in local.listeners : listener.protocol => listener.port...
  }
}
```

The trailing `...` groups duplicates into a list. Without it, duplicate keys are an error. A filter clause keeps some elements: `for port in local.ports : port if port > 100`.

`for_each` on a resource wants a set of strings or a map. A list of objects is the wrong shape until you turn it into a map keyed by a stable name. Keying by list index means a deletion in the middle recreates every later instance. Key by a name the human owns.

Dynamic Blocks: Generating Repetitive Configuration Programmatically

Some resources take repeated nested blocks (`ingress`, `statement`). You cannot put `for_each` on a nested block. `dynamic` generates the blocks from a collection.

```
locals {
  ingress_ports = [22, 443]
}

# Sketch. The security-group type is the provider's, not part of the pet lab.
# resource "aws_security_group" "edge" {
#   dynamic "ingress" {
#     for_each = toset(local.ingress_ports)
#     content {
#       from_port   = ingress.value
#       to_port     = ingress.value
#       protocol    = "tcp"
#       cidr_blocks = ["203.0.113.10/32"]
```

```
#     }
#   }
# }
```

Inside `content`, `ingress.value` is the element and `ingress.key` is the key. Name the dynamic block after the nested block type the provider expects, not after a name you wish the provider had used.

Dynamic blocks hide structure. Three nearly identical blocks written out are easier to review than a dynamic block over a clever local. Use dynamic when the collection is data that changes size. Write the blocks when there are two of them and a human should see both.

Null, Unknown, and Type Conversion: Gotchas Every Engineer Must Know

`null` means "omit this argument" for many optional resource arguments. It does not mean empty string, and it does not mean zero. A default of `""` still sets the argument. A default of `null` leaves it unset so the provider can apply its own default. Those produce different plans.

An unknown value is a value that will exist only after apply: the id of a resource being created in this same plan. Unknown values flow through expressions. They poison anything that must be decided during the plan, especially `count` and `for_each`. You cannot yet know how many instances to plan if the count depends on an id that does not exist.

OpenTofu 1.13 adds two functions aimed at that wall, still settling toward a stable design in 1.14. `convert` asks the planner to treat an unknown as a given type. `assumenotnull` tells the planner a value will not be null. If apply later violates the hint, OpenTofu errors instead of silently doing something else. Use them at a module boundary when a caller passes a value the plan cannot see through. Do not use them to paper over a count that should have been a static map.

`tostring`, `tonumber`, `tolist`, and `tomap` coerce explicitly. Implicit coercion of a bool to a string is how a tag set becomes `"true"` in one environment and `true` in another. Coerce at the edge.

`try(expr, fallback)` catches errors in expressions. It also catches typos. Prefer a type constraint over `try` when the failure is a mistake you want to see.

Summary

- Blocks have a type, labels, and arguments. Names of files do not change that
- Unquoted references are values. Quoted references are strings
- `for` builds collections. `dynamic` builds nested blocks. Stable keys keep state stable
- `null` unsets an optional argument. Unknown values cannot size `count`
- 1.13's `convert` and `assumenotnull` are hints to the planner, checked again at apply

Variables, Locals, and Outputs

Variables, Locals, and Outputs

Variables are the module's arguments. Locals are its private names. Outputs are the only values a parent should read. This chapter is that contract.

Learning goals

- Declare a variable with a type, a default, and a validation
- Load values from a tfvars file without committing a secret
- Use locals to name a repeated expression once
- Pass outputs from a child into a parent without reaching into the child's resources

Input Variables: Types, Defaults, Validation Rules, and Sensitive Values

```
variable "pet_length" {
  description = "Words in the generated name."
  type        = number
  default     = 2

  validation {
    condition     = var.pet_length >= 1 && var.pet_length <= 4
    error_message = "pet_length must be from 1 to 4."
  }
}

variable "name_prefix" {
  description = "Optional prefix. Null means no prefix."
  type        = string
  default     = null
  nullable    = true
}
```

`type` is what stops a string from arriving where a number belongs. `default` makes the variable optional. No default means the caller must supply it. `description` is the documentation. `nullable = false` rejects an explicit null when a default exists and null would be a mistake.

`validation` runs on the value the caller passed. It can see `var.pet_length`. It cannot see other resources' computed attributes in a useful way, and it should not try. Validate shape and policy

here: length, prefix, membership in a list. Validate the live infrastructure with a plan or a test, not with a variable block.

`sensitive = true` marks the value so plans redact it. The value is still in state and still in memory. Sensitivity is a display flag. Chapter 12's rule stands: do not put the long-lived credential in a variable that gets stored at all if you can pass it only to a provider from the environment.

Variable Files (.tfvars): Environment-Specific Configuration Management

`terraform.tfvars` and `*.auto.tfvars` load automatically. Any other name loads with `-var-file=prod.tfvars`. A variable file is assignment, not logic:

```
pet_length = 3
```

Typical layout once you have more than one environment: a root module per environment, each with its own backend and a small tfvars file, rather than one root module and a pile of var-files you forget to pass. Chapter 9 explains why workspaces are a weak substitute for that split. Chapter 14 draws the directories.

Precedence, from lowest to highest: defaults, environment variables `TF_VAR_pet_length`, `terraform.tfvars`, `*.auto.tfvars` in name order, `-var-file`, `-var`. A value supplied later wins. In automation, pass `-var-file` explicitly so the file you reviewed is the file that ran. Do not rely on an `auto.tfvars` nobody mentioned in the pipeline.

Secrets do not go in a tfvars file that is committed. `TF_VAR_state_passphrase` from the pipeline's secret store is the ordinary path. A local `secret.auto.tfvars` that is gitignored works on a laptop and will eventually be copied into a ticket. Prefer the environment.

Locals: Organizing Complex Logic and Avoiding Repetition

Locals are names for expressions. They are not inputs. Callers cannot set them.

```
locals {
  prefix    = var.name_prefix == null ? "" : "${var.name_prefix}-"
  pet_words = var.pet_length
}

resource "random_pet" "service" {
  length = local.pet_words
  prefix = local.prefix == "" ? null : trimsuffix(local.prefix, "-")
}
```

Use a local when the same expression appears twice, or when a conditional is too long to sit inside a resource. Do not use a local as a second, undocumented input. If a caller should change it, it is a variable.

Locals can refer to resources. That reference is unknown until apply when the resource is being created, and everything downstream inherits the unknown. Keep locals that feed `for_each` keys on variables and static data.

Outputs: Exposing Information and Connecting Modules

```
output "name" {
  description = "Generated pet name."
  value       = random_pet.service.id
}

output "state_passphrase_in_use" {
  description = "Never the passphrase. Only whether one was set."
  value       = var.state_passphrase != null
  sensitive   = false
}
```

Outputs are the root module's result (`tofu output`) and the child module's return values (`module.pets.name`). Mark an output `sensitive = true` when its value is sensitive, or OpenTofu will refuse to let a sensitive value flow into it. The flag still does not encrypt the output. Root outputs end up available to anyone who can read the state.

Export the minimum: an id, an ARN, a name. Do not export an entire resource object. Callers will depend on attributes you did not mean to support, and a provider upgrade will break them.

Passing Data Between Modules: The Parent-Child Contract

A parent passes variables. A child returns outputs. That is the whole contract.

```
module "pets" {
  source     = "./modules/pets"
  pet_length = 2
}

output "pet_name" {
  value = module.pets.name
}
```

The parent does not read `module.pets.random_pet.service.id`. That path reaches into the child's resources, couples the parent to names that are none of its business, and breaks the day the child renames a resource. If the parent needs the value, the child grows an output.

The same rule applies sideways. Two root modules do not read each other's variables. They read each other's outputs through remote state (chapter 7), and only the outputs. Remote state is a dependency you should notice in review, because it is an apply-order dependency between projects.

Document the contract next to the variables: units, whether null is allowed, and what the module will create. A variable named `name` that sometimes is a prefix is how two teams apply two different conventions into one account.

Summary

- Variables are the inputs, typed and validated. Sensitive only redacts the UI
- tfvars assign values. The pipeline should name the file it passes
- Locals name expressions. They are not a back door for inputs
- Outputs are the only supported way out of a module
- Parents pass variables and read outputs. They do not reach into child resources

Providers and State

Providers, Resources, and Data Sources in Depth

Providers, Resources, and Data Sources in Depth

A provider is a plugin with credentials. A resource is one object that plugin owns on your behalf. A data source is a question. This chapter is how those three show up for AWS, Azure, and GCP, and how you stop a floating provider version from rewriting the plan.

Learning goals

- Configure a provider without putting a long-lived key in the configuration
- Read a resource schema as required, optional, computed, or deprecated
- Write a small sketch for each of the three large clouds
- Pin provider versions and commit the lock file

How Providers Work: Plugins, Authentication, and Configuration Blocks

Init downloads a binary and records a checksum in `.terraform.lock.hcl`. Plan and apply start that binary and speak the plugin protocol. The provider configuration is the settings every resource of that type inherits:

```
terraform {
  required_version = "~> 1.13.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1"
}
```

Leave the access key out of the block. The AWS provider uses the standard credential chain: environment variables, a shared config file, or a role from the instance or the pipeline. Azure uses a service principal or workload identity. GCP uses application-default credentials or a workload

identity pool. Chapter 12 lists those patterns as the security rule. The configuration above is the whole provider block when the environment already has credentials.

`tofu providers` shows what is required and what was installed. `tofu providers lock` updates the lock file for the platforms you name, so a developer on ARM64 and a pipeline on AMD64 do not fight over checksums. Lock the platforms you actually run.

Resource Arguments: Required, Optional, Computed, and Deprecated

The provider schema, not this book, decides which arguments exist. The categories are stable:

Kind	In configuration	In state after apply
Required	You must set it	Stored
Optional	You may set it. Null usually means "leave the provider default"	Stored if set
Computed	You cannot set it, or you can set a related optional and the provider may override	Filled in by the provider
Optional and computed	If you set it, it is yours. If you do not, the provider chooses and stores the choice	The next plan may want to adopt the provider's value

Computed-only attributes (an instance id, a bucket domain name) are outputs of the create call. Referencing them in the same plan is fine. Using them as a `for_each` key is the unknown-value problem from chapter 4.

Deprecated arguments still work until a provider major version removes them. A plan that warns is a plan you schedule a follow-up for. Ignoring deprecations is how a provider bump becomes a rewrite. Pin the provider, read the changelog at the bump, and do not combine a provider major bump with an OpenTofu minor bump in the same change.

Creating Resources Across Clouds: AWS, Azure, GCP Examples

These are sketches of the same idea: one named object, tags or labels from a variable, no embedded secrets. They will not apply until the matching provider is configured and the account exists. The pet project in chapter 3 is the one that applies as written.

```
resource "aws_s3_bucket" "state" {
  bucket = "example-tofu-state-203-0-113"
}

resource "azurerm_resource_group" "edge" {
  name     = "example-tofu-rg"
  location = "westeurope"
}
```

```
resource "google_storage_bucket" "state" {
  name     = "example-tofu-state-203-0-113"
  location = "EU"
}
```

The shape to copy is not the resource type. It is the discipline around it. The name is explicit, not generated from an unknown. The region is an input. Tags that mark the owning team and the state key are how a human in the console knows the object is managed. Deleting the block later deletes the object. If the object must survive, it does not belong in this configuration, or it belongs behind `prevent_destroy` with a written reason.

On-premises follows the same shape with a different provider: a hypervisor, a DNS server, a load balancer, an IPAM. If the provider is a community plugin, pin it harder and read who publishes it. The registry namespace is the publisher's identity.

Data Sources: Querying Existing Infrastructure Safely

A data source is for an object you do not want this state to own.

```
data "aws_caller_identity" "current" {}

output "account_id" {
  value = data.aws_caller_identity.current.account_id
}
```

Use a data source when another team owns the object and you need its id. Do not use a data source to look up an object this same state created. Use that object's attribute. A data source that searches by a moving filter ("the most recent image") changes plans under you. Pin the identifier when the filter is only a convenience.

A data source that fails because the object is missing fails the plan. That is the right failure when the dependency is real. It is the wrong tool for "create this if it does not exist." Existence-as-create is a resource.

`depends_on` on a data source is occasionally required when the object is created in the same configuration and the data source would otherwise read too early. If you need that, you are usually one refactor away from passing the attribute directly.

Provider Aliases: Managing Multiple Configurations Simultaneously

One default provider configuration is the unnamed block. An alias is a second configuration of the same provider, almost always for a second region or a second account.

```
provider "aws" {
  region = "eu-west-1"
```

```
}

provider "aws" {
  alias  = "backup"
  region = "eu-central-1"
}

resource "aws_s3_bucket" "backup" {
  provider = aws.backup
  bucket   = "example-tofu-backup-203-0-113"
}
```

Resources use the default unless `provider` says otherwise. Modules do not inherit aliases. You pass them:

```
module "backup" {
  source = "./modules/bucket"
  providers = {
    aws = aws.backup
  }
}
```

The child declares `configuration_aliases` only when it must. A module that quietly creates objects in a region the caller did not pass is a bug. OpenTofu 1.9's `provider for_each` is how you build one module instance per region from a map without copying the alias block by hand. The map keys must be static.

Version Constraints: Pinning Providers for Stability

The constraint in `required_providers` is a range. The lock file is the selected build.

Constraint	Meaning
`= 5.70.0`	This build only. Heavy for a library module, right for a root you do not want surprised
`~> 5.70`	Any 5.70.z. A reasonable root-module pin
`~> 5.0`	Any 5.x. Fine for a reusable module that promises compatibility. Loose for a root
`>= 5.0.0`	Anything newer, including 6.0.0, which may break. Avoid in roots

Commit `.terraform.lock.hcl`. A pipeline that runs `tofu init -upgrade` on every apply will take provider upgrades you did not review. Init without `-upgrade` uses the lock. Upgrade in a commit whose diff is the lock file and a plan you read.

Provider major versions are allowed to change arguments and to force resource replacement. Schedule them. The OpenTofu pin and the provider pin move in separate commits so a bad plan has one cause.

Summary

- Providers are plugins. Credentials come from the environment, not from the `.tf` file
- Required, optional, and computed describe who is allowed to set an argument
- AWS, Azure, and GCP sketches share a shape: explicit name, no embedded secret, owned by this state
- Data sources read objects this state does not own
- Aliases are extra configurations you pass into modules on purpose
- The lock file is the pin. The constraint is only the allowed range

State Management and Backends

State Management and Backends

State is the memory of object ids. A backend is where that memory lives and how it is locked. This chapter moves the pet example's local file to a design you can hand to a second operator.

Learning goals

- Say what a state file stores, including values marked sensitive
- Configure a remote backend with a lock
- Read another project's outputs without reading its resources
- Move a state between backends, and recover from a lock that outlived its apply

Understanding State: What It Stores and Why Local Is Not Enough

`terraform.tfstate` is JSON. For each resource address it stores the provider, the type, the id the provider returned, and the attributes from the last refresh. It also stores outputs and a serial number. Sensitive values are stored in full. The sensitive flag changes the CLI, not the file.

Local state fails as soon as two actors exist. Two engineers, or an engineer and a pipeline, apply two copies and each believes it owns the objects. The second apply creates duplicates or destroys the first world's objects because they are absent from the second state. A laptop disk is also a poor backup.

Local state is still correct for a throwaway directory and for learning the commands. The moment the objects cost money or someone else must change them, the state moves.

Backend Types: Local, S3, Azure Blob, GCS, Consul, and More

A backend is declared inside the `terraform` block. It is read at init, and changing it means `tofu init -migrate-state` or a fresh init, not a normal plan.

Backend	Typical use	Locking
`local`	Learning, throwaway	None that helps a team
`s3`	AWS accounts	From OpenTofu 1.10, a lock file in the bucket (`use_lockfile = true`). Older setups use a DynamoDB table

Backend	Typical use	Locking
`azurerm`	Azure	Blob lease
`gcs`	Google Cloud	Object generation / lock in the bucket
`consul`	Environments that already run Consul	Session lock
`http`, `pg`, and others	When the platform standard is something else	Only if that backend implements it. A backend without a lock is a hazard

Pick the backend that sits next to the account you are managing, so the failure domains match a story you can tell: the AWS state lives in an AWS account dedicated to state, not in the same bucket the configuration is free to delete. Access to the state backend is access to every secret the state holds and to the power to unlock or overwrite it. Chapter 12 narrows that IAM.

An example shape for S3, not a bucket this book creates for you:

```
terraform {
  backend "s3" {
    bucket       = "example-tofu-state"
    key          = "pets/terraform.tfstate"
    region       = "eu-west-1"
    encrypt      = true
    use_lockfile = true
  }
}
```

`encrypt = true` is server-side encryption at the bucket. It is not OpenTofu's client-side encryption. You can use both. Client-side encryption is the `encryption` block below. Backend blocks cannot use variables or locals in older OpenTofu. Since 1.8 they can, which is useful and also a way to point production at the wrong bucket with a typo in a local. Prefer a partial backend config (`tofu init -backend-config=backend.hcl`) for the key and bucket, so the committed configuration does not contain an account-specific name you copy between environments by accident.

Remote State Configuration: Locking, Encryption, and Versioning

Locking makes a second apply wait or fail while the first holds the lock. Always turn it on. A backend that cannot lock is the wrong backend for production.

Versioning on the bucket, blob container, or GCS bucket keeps previous state objects. That is your backup when a bad apply writes a bad state. Lifecycle-expire old versions on a schedule you have tested a restore from. A versioned bucket nobody can read is not a backup.

Client-side encryption (OpenTofu 1.7 and later) encrypts the state and, if you ask, the plan, before they leave the process. The backend then stores ciphertext. A passphrase of at least 16 characters, fed from `TF_VAR_`, is the small-team form:

```
variable "state_passphrase" {
  type      = string
  sensitive = true
}

terraform {
  encryption {
    key_provider "pbkdf2" "main" {
      passphrase = var.state_passphrase
    }

    method "aes_gcm" "main" {
      keys = key_provider.pbkdf2.main
    }

    state {
      method   = method.aes_gcm.main
      enforced = true
    }

    plan {
      method   = method.aes_gcm.main
      enforced = true
    }
  }
}
```

`enforced = true` refuses to write plaintext. Lose the passphrase and the state is gone. Store the passphrase in the same secret manager the pipeline uses, and store a recovery copy under the same controls as any other encryption key. For a team, prefer a KMS key provider (`aws_kms`, `gcp_kms`, Azure Key Vault, or OpenBao) so access is IAM, not a shared password. Rotate by writing a new method and a `fallback` block that can still decrypt the old one. Take an unencrypted backup, offline and access-controlled, before the first migration onto encryption.

Reading Remote State: Cross-Referencing Infrastructure Between Projects

`terraform_remote_state` is a data source that reads another state and returns that state's outputs.

```
data "terraform_remote_state" "network" {
  backend = "s3"

  config = {
    bucket = "example-tofu-state"
```

```
    key    = "network/terraform.tfstate"
    region = "eu-west-1"
  }
}

output "vpc_id" {
  value = data.terraform_remote_state.network.outputs.vpc_id
}
```

Only outputs are part of the contract. Reaching for anything else couples you to the other project's resource names. The reader needs permission to read that state, which means the reader can also see every secret in it. Prefer passing an id through a narrow output over sharing a whole state with a wide team. Where the dependency is permanent, a data source on the real object (`data "aws_vpc"`) is often a smaller permission than read access to someone else's state.

Remote state creates an apply order. The network project must have been applied, and its outputs must exist, before the consumer plans. Write that order down. A pipeline that applies consumers in parallel with their producers will fail, or worse, will succeed against stale outputs.

State Migration: Moving Between Backends Safely

Changing the `backend` block and running `tofu init -migrate-state` copies state from the old backend to the new one and offers to delete the old copy. Do this with a lock, with a fresh backup of the source state, and with nobody else applying.

The safe sequence:

1. Stop pipelines that apply this state
2. `tofu state pull > backup.tfstate` and store that file somewhere encrypted, not in git
3. Change the backend configuration in a review
4. `tofu init -migrate-state`
5. `tofu plan` and confirm the plan is empty
6. Re-enable pipelines pointed at the new backend
7. Only then delete the backup, after the retention window you promised yourself

`tofu state mv` moves an address inside one state, or between states with pull and push. Renaming a resource without `moved` or `state mv` destroys and recreates. A `moved` block in the configuration is the reviewable form:

```
moved {
  from = random_pet.service
  to   = random_pet.api
}
```

Leave the block in place through one release so every state has applied it, then remove it.

State Troubleshooting: Corruption, Drift, and Recovery Procedures

A stale lock. The apply died and the lock remains. Read the lock. If the id matches a dead run and no apply is in progress, `tofu force-unlock LOCK_ID` removes it. Unlocking a live apply is how two writers corrupt state. Look at the pipeline before you unlock.

Drift. `tofu plan` shows a change you did not make. Either the console edit is wrong and you apply to put the object back, or the console edit is the new truth and you import it into configuration. Refresh-only (`tofu apply -refresh-only`) updates state to match the world without changing the world. Use it when you have decided the world is right and the state is stale. Do not use it as a weekly habit to hide drift you should have configured.

A corrupt or empty state. Stop. Restore the previous version from the backend's object versioning, or from the backup you pulled. `tofu state push` replaces remote state with a file you have inspected. Pushing the wrong file is a worse incident than the one you were fixing. Inspect with `tofu show -json backup.tfstate` and confirm resource addresses before you push.

Resources in state and not in configuration. The next apply destroys them. That is either the change you wanted or an accidental deletion of a `.tf` file. Read a destroy plan from the bottom. `terraform.tfstate.backup` on a local backend is a single generation. It is not a history.

Never edit state JSON by hand unless you are following a documented recovery and you have a copy of the previous bytes. The CLI subcommands (`state rm`, `state mv`, `import`, `force-unlock`) exist so the serial number and the checksum stay consistent.

Summary

- State stores ids and attributes, including sensitive ones, in full
- A remote backend with a lock and versioning is the minimum for two operators
- Server-side encryption and OpenTofu's client-side encryption solve different problems. Production can use both
- Remote state consumers depend on outputs only
- Migrate with a backup, an empty plan, and pipelines paused
- Unlock only a dead lock. Restore state from a version, do not hand-edit it

Composition

Modules: Building Reusable Infrastructure Components

Modules: Building Reusable Infrastructure Components

A module is a directory of configuration that can be called with inputs and returns outputs. The root module is the one you apply. Everything else is a child. This chapter is how to draw that line so a module stays reusable without becoming a second programming language.

Learning goals

- Call a local child module and read its outputs
- Decide what belongs inside the module and what stays in the root
- Pin a registry or git source the way you pin a provider
- Recognise a module that hides too much

What Is a Module?: Root Modules, Child Modules, and Published Modules

Every directory you run `tofu` in is a root module. It has the backend, the provider configuration, and the state. A child module is a directory you refer to with a `module` block. It has variables, resources, and outputs. It does not have its own backend. Its resources live in the root's state, under addresses like `module.pets.random_pet.service`.

A published module is a child someone else versions. The OpenTofu registry, a git tag, or an OCI artifact are sources. Publishing does not change the language. It changes who is allowed to move the inputs under you.

One root module per state remains the rule. A child that tries to configure a backend is ignored or rejected. If two parts of the system must change on different schedules or different permissions, they are two roots, not two children.

Designing Your First Module: Inputs, Outputs, and Responsibility Boundaries

A module does one thing a caller can name: "a pet name with a prefix", "a bucket with versioning and a policy", "a network with these subnets". If the name needs "and", it is probably two modules.

```
modules/pets/
   versions.tf
   variables.tf
   main.tf
   outputs.tf
```

```
# modules/pets/variables.tf
variable "pet_length" {
  type    = number
  default = 2
}

# modules/pets/main.tf
resource "random_pet" "service" {
  length = var.pet_length
}

# modules/pets/outputs.tf
output "name" {
  value = random_pet.service.id
}
```

The boundary test: a caller who does not care how you implemented it should still be able to use it. If the caller must know you used `random_pet`, the output is leaking the implementation and the module is a folder pretending to be an API. Export the name. Keep the resource type inside.

Do not put provider credentials in a child. The root configures providers and passes aliases. Do not put environment names inside the module as a switch (`if var.env == "prod"`). Pass the values that differ. A module that contains a production special case is a root module that cannot be reused.

Calling Modules: Local Paths, Registry Sources, and Git Repositories

```
module "pets" {
  source     = "./modules/pets"
  pet_length = var.pet_length
}

module "pets_remote" {
  source  = "example.com/team/pets/generic"
  version = "1.4.2"
}
```

A local path (`./modules/pets`) is the right source while the module and the root change together. There is no separate version. The git commit is the version.

A registry source has a `version` constraint. Pin it the way you pin providers. `version = "~> 1.4"` accepts patch releases. Read them. An unbounded version is a supply-chain surprise.

A git source pins a ref:

```
module "pets_git" {
  source = "git::https://example.invalid/team/pets.git?ref=v1.4.2"
}
```

Pin a tag or a commit. Pinning a branch tracks whatever the branch points at the next time someone deletes `.terraform/modules` and inits. That is not reproducibility. Private git needs the pipeline's credentials, not a token in the source URL.

OpenTofu 1.8 can use variables in module `source` in more cases than Terraform 1.5 could. A source that changes with a variable changes which code you are about to apply. Keep sources static in root modules unless you have a reviewed reason.

The OpenTofu Registry: Publishing and Consuming Community Modules

`registry.opentofu.org` serves provider and module metadata. Consuming a module is the `source` and `version` above, then `tofu init`. The registry records who published the module. That namespace is the identity you are trusting, the same way a provider namespace is.

Publishing is a review, not an upload button hidden in the CLI. The registry project accepts modules through its documented submission process. A module you publish is a contract: input types, outputs, and the objects you create. A breaking change is a major version. Yanking a version that someone has pinned breaks their init. Do not republish a tag with different code.

Prefer a small number of modules your team owns over a deep tree of community modules you have not read. A community module that creates IAM is a module you read line by line before the first apply. Fork and pin if you depend on it and you cannot accept surprise commits.

Advanced Module Patterns: Stacks, Composition, and Versioned Releases

Composition means a root module calls a few child modules and passes outputs across. That is the pattern to keep.

```
module "name" {
  source     = "./modules/pets"
  pet_length = 2
}

module "record" {
  source = "./modules/record"
```

```
  name   = module.name.name
}
```

A stack, in the sense that matters here, is that root plus its state plus its pipeline. It is not a new language. HashiCorp's Terraform Stacks product is a separate commercial workflow. You do not need it to compose OpenTofu modules.

Versioned releases apply to modules other teams call. Tag semver. A changelog says whether inputs were added (compatible) or renamed (breaking). Internal modules used by one root can live in the same repository and move without tags until a second caller appears. The day a second caller appears, the module gets a version or it gets copied. A floating local path shared by two roots via a relative `../../` is how unrelated applies break together.

`for_each` on a module creates one instance per key, each with the same child code and different inputs. The keys are stable names. This is the right way to stamp out one bucket per application. It is the wrong way to hide a loop the reader cannot see without expanding state.

Anti-Patterns: Monolithic Modules, Hidden Dependencies, and Over-Abstraction

The monolith. One module creates the network, the cluster, the database, and the DNS. Every change plans everything. Split along change rate and along ownership.

Hidden dependencies. A module looks up a VPC by tag inside a data source the caller did not pass. The caller cannot see the dependency. Pass the VPC id in. The plan then shows where it came from.

Over-abstraction. A "universal cloud instance" module with `var.cloud` and three implementations is three modules and a switch that fails in the ways all three clouds fail. Write the AWS module and the Azure module. Share a naming local if you must. Do not share a resource graph that does not exist.

Pass-through variables. A child variable that exists only to be passed to a grandchild, unchanged, with no default and no validation, means the middle module has no opinion. Delete the middle module.

Defaults that create cost. A default of `true` on "create a NAT gateway per zone" surprises a caller in a lab account. Defaults should be the safe, small choice. Production opts in.

Summary

- The root owns the backend and the state. Children own a reusable graph
- Inputs and outputs are the API. Resource addresses are not
- Pin registry versions and git refs. A branch is not a pin
- Compose a few modules at the root. Do not build a framework
- A module that looks up its own dependencies hides the thing the plan should show

Refactoring Without Replacement: moved, removed and Deprecations

Refactoring Without Replacement: moved, removed and Deprecations

OpenTofu matches configuration to state **by address**. Rename `aws_instance.web` to `aws_instance.frontend` and, as far as the plan is concerned, one object vanished from the configuration and a new one appeared. The plan therefore says *destroy one, create one*. For a stateless VM that is an outage. For a database or a bucket it is data loss with a green checkmark.

This chapter covers the three declarative tools that let a configuration change shape without touching what it manages. `moved` changes an address. `removed` stops managing an object without destroying it. `deprecated` warns module callers before a breaking change lands.

> **What was and was not run**
>
> No `tofu` binary was available on the authoring box and installs were not allowed, so no plans were executed. Every block below follows the current OpenTofu documentation for the 1.13 series. Plan wording is described, not captured. Before using a pattern in production, run `tofu plan` and confirm the plan contains no destroys.

Learning goals

- Write `moved` blocks for renames, `count`/`for_each` changes, module renames and module splits
- Know when a whole-resource move applies to every instance and when an instance key makes it instance-specific
- Retire a resource from management with `removed` and `destroy = false`, and run a cleanup provisioner with `destroy = true`
- Plan the lifetime of these blocks in shared modules, and warn callers with `deprecated`

`moved`: change the address, keep the object

```
resource "aws_instance" "b" {
  count = 2
  # ...
}

moved {
  from = aws_instance.a
  to   = aws_instance.b
```

```
}
```

Before planning `aws_instance.b`, OpenTofu looks for an object at `aws_instance.a` in state. If it finds one, it re-addresses the object and plans as if it had always been `b`. Both addresses here name the *whole resource*, so `a[0]` becomes `b[0]` and `a[1]` becomes `b[1]` without listing them. A configuration that never had `a` (a fresh environment using the same module) simply ignores the block.

Changing how many: instance keys

Once either address carries an instance key, the move is about **specific instances**:

```
# single resource -> for_each: keep the old object as "small"
moved {
  from = aws_instance.a
  to   = aws_instance.a["small"]
}

# for_each key renamed
moved {
  from = aws_instance.b["small"]
  to   = aws_instance.b["tiny"]
}

# count -> for_each, one block per index
moved {
  from = aws_instance.c[0]
  to   = aws_instance.c["small"]
}
moved {
  from = aws_instance.c[1]
  to   = aws_instance.c["tiny"]
}

# count -> single, keeping index 2
moved {
  from = aws_instance.d[2]
  to   = aws_instance.d
}
```

Adding `count` to a resource that had none makes OpenTofu move the existing object to `[0]` automatically, unless a `moved` block mentions that resource. Write the block anyway. The next reader sees intent instead of an implicit rule.

The common trap is switching `count` to `for_each` over the same list. Without `moved` blocks, every instance is destroyed and recreated, because `[0]` and `["web"]` are different addresses.

Modules

Renaming a module call moves everything under it:

```
module "b" {
  source = "../modules/example"
}

moved {
  from = module.a
  to   = module.b
}
```

`module.a.aws_instance.example` becomes `module.b.aws_instance.example`, for every instance if the call used `count` or `for_each`. Adding `count` to a module call works like adding it to a resource. Pick the key that inherits the existing objects:

```
module "a" {
  source = "../modules/example"
  count  = 3
}

moved {
  from = module.a
  to   = module.a[2]   # existing objects become instance 2; 0 and 1 are created
}
```

Splitting a module

A module that grew too big can be split into `x` and `y` while existing users upgrade in place. The original module becomes a **shim**: it calls the two new modules and records where each resource went.

```
# the original module, now a shim
module "x" {
  source = "../modules/x"   # holds aws_instance.a and .b
}

module "y" {
  source = "../modules/y"   # holds aws_instance.c
}

moved {
  from = aws_instance.a
  to   = module.x.aws_instance.a
}
```

```
moved {
  from = aws_instance.b
  to   = module.x.aws_instance.b
}
moved {
  from = aws_instance.c
  to   = module.y.aws_instance.c
}
```

Addresses in `moved` are resolved **relative to the module that declares them**, and a module may only make statements about its own objects and its children's. So this breaks the usual "child module is a closed box" rule, and only makes sense when one team ships all three modules together. If the target call uses `count` or `for_each`, name the instance: `to = module.new[2].aws_instance.example`.

What `moved` cannot do

- Turn a managed `resource` into a `data` source.
- Move between resource *types* unless the provider supports migrating from the old type. Check its documentation.
- Apply to `ephemeral` blocks, which are never in state.

The lifetime of a `moved` block

Removing a `moved` block is a **breaking change**. Any state still at the old address will plan a destroy. In a module other people consume, keep historical moves indefinitely. When an object moves twice, chain the blocks so both kinds of user can upgrade:

```
moved {
  from = aws_instance.a
  to   = aws_instance.b
}
moved {
  from = aws_instance.b
  to   = aws_instance.c
}
```

In a private root configuration with a known set of states, you may delete the blocks once every state has applied the version that contains them.

`removed`: stop managing, keep the object

Deleting a `resource` block plans a destroy. To hand an object over to another configuration, another tool or a human, delete the block and replace it with:

```
removed {
  from = aws_instance.web
  lifecycle {
    destroy = false
  }
}
```

The plan then shows the object being removed from state with nothing destroyed, sometimes summarised as "to forget". Rules:

- `from` names the resource without instance keys. `aws_instance.web[0]` is invalid.
- `lifecycle` is optional, but OpenTofu warns when it is missing. State your intent.
- `destroy = true` behaves exactly like deleting the block, and keeps a written record that the resource once existed.

With `destroy = true`, a `removed` block can carry a destroy-time provisioner, so cleanup that lived on the old resource still runs:

```
removed {
  from = aws_s3_bucket.example
  lifecycle {
    destroy = true
  }
  provisioner "local-exec" {
    when    = destroy
    command = "echo 'destroying bucket ${self.bucket}'"
  }
}
```

The provisioner runs only when both `destroy = true` and `when = destroy` are set. It can reference only `self`, `each.key` and `count.index`, which are what state still knows about the object.

`removed` is the reviewable, plan-visible form of `tofu state rm`. Use it in shared code and in pipelines. Keep the CLI for emergencies.

Handover recipe: moving an object to another configuration

1. In the receiving configuration, add the `resource` block and an `import` block for the object. Plan: one import, no changes.
2. Apply the receiving side.
3. In the giving configuration, replace the `resource` block with `removed { destroy = false }`. Plan: one removed from state, nothing destroyed.
4. Apply the giving side. Delete the `removed` and `import` blocks in a later change.

Applying them in the other order leaves a window where nothing manages the object. Leaving out step 3 leaves two configurations fighting over it.

`deprecated`: warn callers first

Refactoring a module's *interface* (renaming an input or an output) needs action from callers. Variables and outputs take a `deprecated` attribute with a migration hint:

```
variable "instance_type" {
  type       = string
  default    = null
  deprecated = "Use var.size instead; instance_type will be removed in v3."
}

output "public_ip" {
  value      = aws_instance.web.public_ip
  deprecated = "Use the public_ips output (a map keyed by instance)."
}
```

Callers that set the variable or read the output get a warning. Output deprecation warnings appear on `tofu plan` and `tofu apply`, **not** on `tofu validate`, so CI that only validates won't show them.

Choosing the tool

You want to	Use	Not
Rename a resource or module call	`moved`	`tofu state mv` in a pipeline
Switch `count` to `for_each`	`moved` per instance	Hoping keys match
Split a module	Shim module plus `moved`	Asking every caller to run state commands
Stop managing but keep	`removed` with `destroy = false`	Deleting the block
Delete, with cleanup	`removed` with `destroy = true` and a provisioner	Leaving a provisioner on a dead resource
Rename a module input or output	`deprecated` for one release, then remove	A silent breaking change

Diagnosis

Plan shows	Likely cause	Fix
`-/+` or destroy plus create after a rename	No `moved` block, or `from` typo	Add or fix `moved` and re-plan until it reports "has moved to"
Every instance replaced after a `count` to `for_each` change	Instance-level moves missing	One `moved` per index-to-key mapping
A destroy after deleting old `moved` blocks	Some state was still at the old address	Restore the blocks
Error on `removed` with `[0]`	Instance key in `from`	Use the bare resource address
Warning: `removed` without `lifecycle`	Intent not stated	Add `lifecycle { destroy = ... }`
No deprecation warnings in CI	Only `tofu validate` runs	Run `tofu plan` in CI

Operational care

- Make "no destroys" a review gate for refactor PRs. `tofu show -json` on a saved plan makes it scriptable.
- Put `prevent_destroy` on stateful resources. It turns a missed `moved` into an error instead of an outage.
- Keep refactors separate from functional changes so the plan diff is only address moves.
- For shared modules, record moves in the changelog and keep the blocks. For root configurations, put the cleanup PR on a calendar.

Practice

1. A root module has `aws_instance.app` with `count = 3`. You want `for_each` over `["blue", "green", "canary"]`. Write the `moved` blocks.
2. You're moving `aws_route53_zone.main` from a monolith into a dedicated DNS configuration. Write both sides of the handover and give the apply order.
3. Your published module renames output `vpc` to `network`. Plan the deprecation over two releases.

Recall card

- State is keyed by address. Any rename without `moved` is destroy plus create.

- A move without instance keys covers all instances. A move with a key covers that instance only.
- `moved` resolves relative to its module and is a permanent upgrade path in shared modules. Chain repeated moves.
- `removed { lifecycle { destroy = false } }` forgets without destroying. No instance keys. A destroy-time provisioner runs only with `destroy = true`.
- `deprecated` on variables and outputs warns callers. Output warnings appear on plan and apply, not validate.

Sources

- OpenTofu, Refactoring
- OpenTofu, Resource blocks: Removing Resources
- OpenTofu, Input variables

Workspaces, Lifecycle Rules, and Dependencies

Workspaces, Lifecycle Rules, and Dependencies

These are the controls that change what a plan is allowed to do: which state you are in, whether a resource may be destroyed, and which edges the graph waits for. They are also easy to use as a substitute for a layout you should have drawn in chapter 14.

Learning goals

- Explain a workspace as a separate state, not a separate configuration
- Decide when a workspace is the wrong environment boundary
- Set `create_before_destroy`, `prevent_destroy`, and `ignore_changes` for a reason
- Read a dependency graph and force replacement without editing the cloud by hand

Workspaces: Multiple Environments from One Configuration

A workspace is a named state inside one backend, selected with `tofu workspace select`. The configuration does not change when you switch. `terraform.workspace` (the name of the value, kept for compatibility) is a string you can interpolate, usually into names.

```
resource "random_pet" "service" {
  length = 2
  prefix = terraform.workspace
}
```

`default` exists even when you have never created another workspace. `tofu workspace new staging` makes a second state. Apply in `staging` does not update `default`. That isolation is the feature.

The CLI remembers the selected workspace in local files. A pipeline must select the workspace explicitly (`tofu workspace select production`) and fail if the select fails. An apply that runs in `default` because the select was skipped is an apply into the wrong state.

When Workspaces Help (and When They Hurt): Practical Guidance

Workspaces help when the environments are the same shape, the same permissions boundary, and the same blast radius: a developer sandbox per person, or a short-lived test state. They hurt when production and development differ in account, credentials, or who is allowed to apply.

Production state in the same backend configuration as development, separated only by a workspace name, means a missed `workspace select` can plan against production with development credentials or the reverse. Separate root modules, separate backend keys, and separate pipelines make that mistake a file you did not mean to enter rather than a flag you forgot.

Do not use the workspace name as the only difference in a large conditional (`terraform.workspace == "production" ? big : small`) that changes counts and providers. That configuration cannot be read without knowing which state it will meet. Two directories that call the same child module with different variables are longer and clearer.

If you already have workspaces and they are small, keep them until the permission boundary appears. Do not migrate in the middle of an incident.

Lifecycle Arguments: create_before_destroy, prevent_destroy, ignore_changes

`lifecycle` is a meta-argument on a resource.

`create_before_destroy = true` creates the replacement before deleting the old object. Use it when the name can differ (so both can exist) and something else will retarget, such as a listener that follows the new instance. It fails when the identity must be unique, such as a fixed bucket name. The plan will tell you the create cannot proceed while the old object holds the name.

`prevent_destroy = true` makes any plan that would destroy the resource an error. Use it on the state bucket's own configuration if that configuration lives in OpenTofu at all, and on data you cannot recreate. It also makes refactors louder, which is what you want. Removing the flag is a reviewed change, not a quick comment-out during an apply.

`ignore_changes` tells the plan to leave listed arguments alone even if the configuration and the world differ.

```
resource "random_pet" "service" {
  length = 2

  lifecycle {
    ignore_changes = [separator]
  }
}
```

Ignore a field when something outside OpenTofu is the writer of that field and you have accepted the split: an autoscaler setting capacity, a security tool setting a tag. Ignoring `all` with `ignore_changes = all` hides every drift on that resource. That is rarely what you meant. Ignoring an argument you set yourself hides your own edits. Take the argument out of the configuration if OpenTofu should not own it.

`replace_triggered_by` forces replacement when another value changes. Use it when a resource does not see an argument change that should have rebuilt it. Prefer fixing the argument over a trigger when the provider supports the change.

Dependency Management: Explicit depends_on vs. Implicit References

A reference is a dependency. If `module.record` takes `module.name.name`, the record waits for the name. You do not add `depends_on` for that edge. The graph already has it.

`depends_on` is for edges the configuration cannot see: a data source that reads an object created in the same apply, or a provisioner-shaped side effect you have not yet removed. Every explicit edge is a serialisation. A module-level `depends_on` waits for the entire module, which is coarser than waiting for one output and can slow the plan's apply for no gain.

Do not `depends_on` a resource to paper over a missing attribute reference. Pass the attribute. The next reader can then see why the order exists.

Cycles are errors. A cycle means two objects each need the other's id. Break it by creating one of them with a name you already know, or by splitting the states. A cycle is a design signal, not a flag to relax.

The Dependency Graph: Visualizing and Debugging Execution Order

`tofu graph` prints a DOT graph of the addresses and their edges. Render it when you do not believe the order an apply is taking. Most of the time the plan's explanation of a replacement is enough and the full graph is noise.

What you are looking for:

- An edge you did not intend, usually from a reference used only in a tag
- A missing edge, usually a data source that reads too early
- A module that everything depends on because its outputs are a grab-bag

`tofu apply -target=random_pet.service` applies only that node and what it requires. Targeting is a recovery tool. An apply that always targets has left the declarative model, and the untargeted objects drift from the configuration without anyone noticing. Follow a targeted apply with a full plan the same day.

Tainting Resources: Forcing Recreation When Needed

Tainting marks an object so the next plan replaces it, even though the configuration did not change. The modern form is:

```
tofu apply -replace=random_pet.service
```

`-replace` is the flag to use. The older `tofu taint` still exists and is easier to forget in state. Replacement runs the destroy and create according to `create_before_destroy` and the graph. For the pet resource that only changes a name. For a database it deletes data. Read the plan.

Replace when the object is wrong in a way the configuration cannot express: a virtual machine whose user-data ran once and failed, a secret that was issued and must be reissued. Do not replace

as a substitute for understanding a drift. If the configuration is wrong, fix the configuration. If the world is wrong and the configuration is right, replace.

Summary

- A workspace is another state for the same configuration. Select it explicitly
- Separate roots when accounts, credentials, or blast radius differ
- `prevent_destroy` and `create_before_destroy` are promises about replacement. `ignore_changes` is a list, not `all`
- References build the graph. `depends_on` is for edges a reference cannot see
- `-replace` forces recreation. `-target` is a recovery hatch followed by a full plan

Provider Instances with for_each and the enabled Meta-Argument

Provider Instances with for_each and the enabled Meta-Argument

Chapter 6 introduced aliases: a second, hand-written provider block for a second region or account. Chapter 8 used `for_each` to stamp out module instances. This chapter joins the two. Since OpenTofu 1.9 a provider block can itself take `for_each`, so the set of regions becomes data instead of copy-pasted blocks. Since 1.11 the `enabled` meta-argument replaces the `count = var.x ? 1 : 0` trick for "zero or one" resources. Both are on the pinned 1.13.2, and both have one rule you must learn before production: what happens when something is *removed*.

> **What was and was not run**
>
> No `tofu` binary was available on the authoring box and the run rules forbid installs, so nothing here was executed. The HCL follows the OpenTofu 1.13 documentation. The warning and error texts are quoted from that documentation, not captured locally. The AWS blocks are sketches; validate them with `tofu validate` before use.

Learning goals

- Declare many instances of one provider configuration from a map or set
- Bind resources and whole modules to a chosen provider instance
- Retire a region without a "provider instance not present" error
- Use `lifecycle { enabled = ... }` for optional single resources and migrate from `count`

Static aliases do not scale

With aliases, three regions mean three provider blocks, and every module call is written three times. Adding a region is a code change in several places, and forgetting one is silent. What you want is a variable:

```
variable "aws_regions" {
  type = map(object({
    vpc_cidr_block = string
  }))
}

provider "aws" {
  alias    = "by_region"
```

```
  for_each = var.aws_regions
  region   = each.key
}
```

```
# terraform.tfvars
aws_regions = {
  eu-central-2   = { vpc_cidr_block = "10.1.0.0/16" }
  ap-northeast-1 = { vpc_cidr_block = "10.2.0.0/16" }
}
```

The rules for the provider block:

- `for_each` is allowed only together with `alias`. The *default* configuration of a provider must always be exactly one instance, so OpenTofu can pick it automatically.
- The value must be a map, an object, or a set of strings. Map keys (or set elements) become **instance keys**.
- Provider configuration is evaluated before anything is applied, so the collection must be known at plan time: input variables and literals, not attributes of resources created in the same run.

Selecting an instance

A provider reference is not an ordinary expression. It may appear only in a resource's or data source's `provider` argument, or in a module's `providers` map. With `for_each`, the reference grows an index:

```
resource "aws_vpc" "private" {
  for_each = {
    for region, config in var.aws_regions : region => config
    if config != null
  }
  provider   = aws.by_region[each.key]
  cidr_block = each.value.vpc_cidr_block
}
```

The configuration name (`aws.by_region`) is static so OpenTofu can draw the dependency graph before evaluating anything. Only the key in brackets is dynamic. Consequently every instance of one resource block binds to *the same* provider block, though each may bind to a different instance of it.

One module instance per region

The more common shape is a whole module per region. The child module uses only its *default* AWS provider and knows nothing about regions:

```
module "per_region" {
  source   = "./per-region"
  for_each = {
    for region, config in var.aws_regions : region => config
    if config != null
  }

  providers = {
    aws = aws.by_region[each.key]
  }

  region_name    = each.key
  vpc_cidr_block = each.value.vpc_cidr_block
}
```

This is the payoff. The child stays simple and reusable, the root decides where it runs, and adding a region is one entry in `terraform.tfvars`.

The removal rule

OpenTofu uses a resource's provider instance for *every* action on it, including destroying it after it disappears from configuration. So a provider instance must outlive all of its resources by at least one plan/apply round.

If the provider and the resource share the same `for_each` collection, deleting a key removes both at once, and the plan fails:

```
  Error: Provider instance not present

  To work with aws_cloudwatch_log_group.lambda_cloudfront["sa-east-1"]
  its original provider instance at
  provider["registry.opentofu.org/hashicorp/aws"].by_region["sa-east-1"]
  is required, but it has been removed.
```

OpenTofu warns you earlier, at the moment you write it, with *"Provider configuration for_each matches resource"*. Treat that warning as a design error. Two patterns avoid it.

Null means retiring. The examples above filter out `null` values for the resource but not for the provider. Setting a region's value to `null` keeps its provider instance while the plan destroys its resources. Delete the key on the next round.

An explicit disabled set. Clearer to reviewers:

```
variable "aws_active_regions" {
  type    = set(string)
  default = ["us-east-1", "sa-east-1"]
}
```

```hcl
variable "aws_disabled_regions" {
  description = "Regions whose resources should be destroyed."
  type        = set(string)
  default     = []
}

provider "aws" {
  alias    = "by_region"
  for_each = var.aws_active_regions # superset
  region   = each.key
}

resource "aws_cloudwatch_log_group" "lambda_cloudfront" {
  for_each = setsubtract(var.aws_active_regions, var.aws_disabled_regions)
  provider = aws.by_region[each.key]
  name     = "/aws/lambda/${each.key}.lambda"
}
```

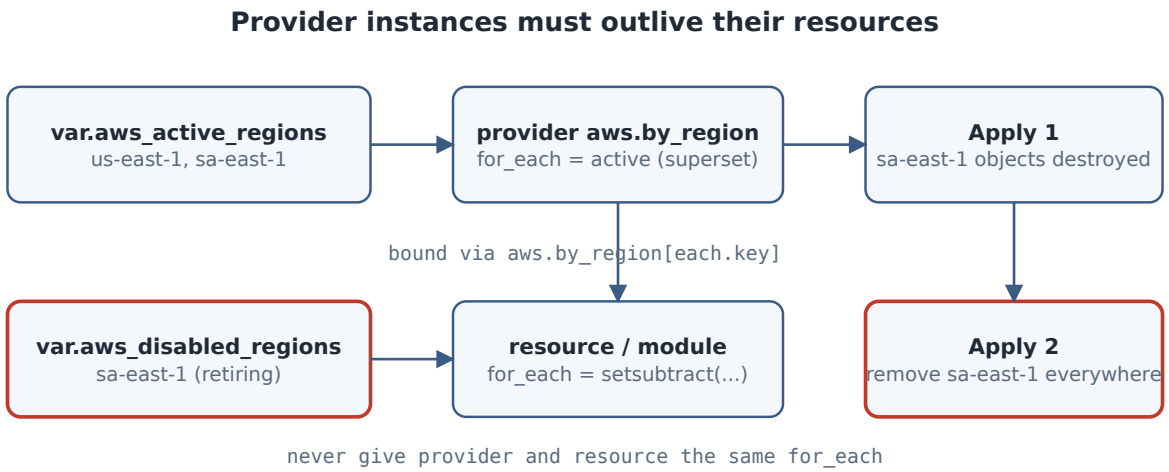

Figure 1: Retiring a region in two applies: the provider iterates the superset, resources the subset.

Retiring `sa-east-1` is then two reviewed changes: add it to `aws_disabled_regions` and apply (resources destroyed, provider kept), then remove it from both sets and apply again. If you are already stuck with a shared collection, the escape hatch the warning suggests is a targeted destroy first: `tofu apply -destroy -target=...` for the affected addresses, then remove the key.

`enabled`: zero or one, without `[0]`

The other half of "how many of these" is the optional single resource. The old idiom was:

```
resource "aws_instance" "bastion" {
  count = var.create_bastion ? 1 : 0
  # ...
}

output "bastion_id" {
  value = length(aws_instance.bastion) > 0 ? aws_instance.bastion[0].id : null
}
```

It works, but every reference carries `[0]`, and the address in state is `aws_instance.bastion[0]`. OpenTofu 1.11 added `enabled` inside `lifecycle`:

```
variable "create_bastion" {
  type    = bool
  default = false
}

resource "aws_instance" "bastion" {
  # ...
  lifecycle {
    enabled = var.create_bastion
  }
}

output "bastion_id" {
  value = aws_instance.bastion != null ? aws_instance.bastion.id : null
}
```

When `enabled` is `false`, the resource is not created and the expression `aws_instance.bastion` evaluates to `null`. Reading an attribute of it directly is an error ("Attempt to get attribute from null value"), so guard with a null check or `try()`. The docs note that `try()` is concise but also masks unrelated errors, so prefer the explicit null check in modules other people consume.

`enabled` works on `resource` and `module` blocks (and on ephemeral resources). Its restrictions:

- It cannot be combined with `count` or `for_each` on the same block.
- Its value must be a known, non-null, non-sensitive, non-ephemeral boolean at plan time.

Migrating from count

Replace `count = var.x ? 1 : 0` with `lifecycle { enabled = var.x }` and drop the `[0]` from references. On the next apply OpenTofu moves the state entry from `aws_instance.bastion[0]` to `aws_instance.bastion` by itself. No `moved` block and no `tofu state mv` are needed. Read the plan anyway: it should show a move, not a destroy and create.

Choosing the construct

Need	Use
Optional single resource or module	`lifecycle { enabled = ... }`
Several numbered, interchangeable copies	`count`
Several named copies	`for_each`
Same infrastructure in a dynamic set of regions or accounts	provider `for_each` plus module `for_each`, with a superset provider collection
Fixed second region, never changes	a plain alias (chapter 6)

Diagnosis

Symptom	Cause	Fix
"Provider configuration for_each matches resource" warning	Same collection on provider and resource/module	Give the provider a superset; filter the resource
"Provider instance not present" error	Key removed while resources still exist	Re-add the key, retire via null or the disabled set, then remove
Error that `for_each` is only valid with `alias`	`for_each` on a default provider block	Add `alias` and pass instances explicitly
Provider `for_each` value unknown	Collection derives from a resource attribute	Drive it from variables or literals
"Attempt to get attribute from null value"	Reading a disabled resource directly	Null check or `try()`
Plan wants to destroy and recreate after switching to `enabled`	Running a version older than 1.11, or `count` still present	Check `tofu version`; remove `count`

Operational care

- Keep the region map in one tfvars file per root and review changes to it like code. It now decides where infrastructure exists.
- Each provider instance is a separate plugin configuration, with its own API calls on every plan. Dozens of regions or accounts make plans slower; split roots before that becomes painful.
- Document in module READMEs that the module expects only a default provider, so callers know to pass `aws = aws.by_region[each.key]`.

Practice

1. Convert a configuration with three hand-written regional aliases to a single `aws.by_region` provider with a map variable and a per-region module.
2. Retire one region using the disabled-set pattern and write down the two plans you expect, before you run them.
3. Find a `count = ... ? 1 : 0` resource in your code base. Migrate it to `enabled` and confirm the plan shows a move.

Recall card

- Provider `for_each` requires `alias`; keys must be known at plan time.
- Bind with `aws.by_region[each.key]`; the block name is static, the key is dynamic.
- The provider collection must be a superset of its resources' collections. Retire in two applies.
- `lifecycle { enabled = bool }` gives zero-or-one without `[0]`, and migrates from `count` automatically.

Sources

- OpenTofu, Provider configuration: `for_each`
- OpenTofu, The `providers` meta-argument: module instances with differing provider instances
- OpenTofu, The `enabled` meta-argument

Provisioners and Importing Existing Infrastructure

Provisioners and Importing Existing Infrastructure

Provisioners run a script against a resource after OpenTofu has created it. Import does the opposite kind of work: it tells state about an object that already exists. One of these belongs in a modern module. This chapter is how to tell them apart, and how to bring a legacy object under management without recreating it.

Learning goals

- Explain why a provisioner breaks the declarative model
- Name the few cases where `local-exec` is still honest
- Import an existing object and generate configuration for it
- Plan a migration that ends with OpenTofu as the owner

The Problem With Provisioners: Why They Are Discouraged

A resource argument is stored in state and corrected on the next plan. A provisioner runs and is forgotten. The script's effect is invisible to the next plan unless the script's failure happened to fail the apply. The object looks clean while the package the script installed has drifted.

Provisioners also put an imperative tool inside a declarative run: SSH keys, WinRM (removed from OpenTofu 1.13's `file` and `remote-exec`), a shell on the machine running `tofu`. That shell has the pipeline's credentials. A `local-exec` that curl-posts to an API is a second provider with none of the provider's diff.

The replacements are ordinary provider features. Cloud-init and user-data are arguments. Configuration management is a tool that converges on its own. A Kubernetes manifest is a resource of its provider. An API call you need every time is a provider resource, even if you have to write a small one. Reach for a provisioner only after those options are genuinely absent.

When Provisioners Make Sense: file, local-exec, remote-exec, and Beyond

`local-exec` runs on the machine where `tofu` runs. The honest uses are narrow: call a control-plane API the provider does not cover yet, during a migration you have dated; or render a one-shot artifact the provider cannot. Even then, a `null` resource with a `local-exec` is a resource whose only behaviour is the script, and it will not re-run when the script's inputs change unless you wire a trigger.

`file` and `remote-exec` copy and run on the new machine over SSH. They need a connection block, a reachable address, and a key. They fail when the network path from the pipeline to the instance does not exist, which is the common production case. In 1.13 they do not speak WinRM. If the instance is Windows, use a different channel or, better, an image that does not need a post-create script.

`creation-time` provisioners run on create. `destroy-time` provisioners run before destroy. A destroy-time provisioner that must succeed or it blocks destruction will block an incident. Prefer the provider's own deletion behaviour.

There is no "beyond" that makes provisioners a design. Chef and Puppet provisioner bridges were the old pattern. Leave them in the history of the module you are migrating.

Best Practices If You Must Use Provisioners: Idempotency and Error Handling

If a migration still has one:

- Make the script idempotent. Running it twice converges. A script that appends a line every time is not finished
- Fail the script with a non-zero status. A provisioner that swallows errors reports a successful apply
- Pass data as arguments or environment, not as a string interpolated into a shell command you did not quote
- Trigger replacement or re-run from a hash of the inputs, or accept that the script runs only on create and document that
- Keep the script in the repository, not on a laptop path
- Delete the provisioner in a follow-up commit once the provider has the resource. Put that follow-up on the same change request

`on_failure = continue` hides the failure. Do not set it to get a green pipeline.

Importing Existing Resources: The import Command and Workflow

Import writes an existing remote object into state under an address. It does not write configuration. After a successful import, a plan with no configuration for that address wants to destroy the object. Write the configuration first, or generate it in the same step, and plan before you apply.

The declarative form, stable since the 1.5 language:

```
import {
  to = aws_s3_bucket.state
  id = "example-tofu-state-203-0-113"
}

resource "aws_s3_bucket" "state" {
```

```
  bucket = "example-tofu-state-203-0-113"
}
```

`tofu plan` shows an import. `tofu apply` records the object in state. The next plan should be empty. If it is not, the configuration does not match the object and apply would change a legacy bucket. Fix the configuration until the plan is empty. That empty plan is the proof you imported rather than adopted-and-mutated.

The one-shot CLI `tofu import aws_s3_bucket.state example-tofu-state-203-0-113` does the same state write without an `import` block. Prefer the block. It is reviewed, it can take `for_each`, and it documents the migration in git. Remove the `import` block after every state has applied it, the same way you remove a `moved` block.

The `id` string is provider-specific. For many resources it is the cloud id. For others it is a composite (`name/namespace`). Read the provider's import document for that resource type. Guessing the id imports the wrong object or imports nothing.

Generating Configuration for Imported Resources: State Inspection Techniques

`tofu plan -generate-config-out=generated.tf` writes HCL for addresses in `import` blocks that do not yet have configuration. The file is a starting point. It is not a module.

Generated configuration dumps every attribute the provider returned, including ones you should not set because they are computed, and ones that differ from the provider's defaults in ways that will churn. Edit it down to the arguments you intend to own. Then plan again.

Inspection tools while you edit:

Command	What you learn
`tofu state show aws_s3_bucket.state`	Attributes currently stored
`tofu state list`	Addresses in this state
`tofu show -json`	The same facts for a script
A refresh-only plan	Whether the world and the state still agree

Do not generate configuration for an entire account in one pass. Import the object you mean to change, get an empty plan, and then import the next one. A generated file of hundreds of resources is a state you cannot review.

Migration Strategies: Bringing Legacy Infrastructure Under OpenTofu Control

A migration is a sequence of empty plans, not a flag day.

1. Inventory the objects and decide which ones OpenTofu will own. Some should stay outside: the state bucket itself is often created by hand or by a tiny bootstrap root so the backend exists before init
2. Write or generate configuration for one group that changes together
3. Import, plan, and stop when the plan is empty
4. Apply a no-op so state and configuration are known-good
5. Only then change an argument in a later commit
6. Remove the `import` blocks
7. Turn on `prevent_destroy` for objects the migration must not casually delete

Two teams must not manage the same object. If a console workflow still edits it, either stop that workflow or `ignore_changes` the specific argument and say why. Dual ownership is the drift you will be paged for.

Account-level tools that only scan and suggest imports are fine as a map. They are a poor author of the module boundary. You still decide what the root module is.

When the legacy object cannot be imported because the provider has no import for that type, you are stuck with a one-time recreation or with leaving it unmanaged. Recreate only when you can afford the replacement. Write that down. A provisioner is not the bridge across a missing import.

Summary

- Provisioners run and forget. Resource arguments are corrected on the next plan
- OpenTofu 1.13's file and remote-exec provisioners do not speak WinRM
- If a script remains, it is idempotent, it fails loudly, and it has a removal date
- Import with a reviewed `import` block, then edit configuration until the plan is empty
- Generated HCL is a draft. Migrate one group at a time, and do not share ownership with the console

Confidence

Testing, Validation, Formatting, and Policy Enforcement

Testing, Validation, Formatting, and Policy Enforcement

A plan a human reads is the most important test. The commands in this chapter exist so that human never has to spend the review on indentation, a missing variable, or a security group open to the world. Sentinel is not one of those commands on OpenTofu.

Learning goals

- Run `tofu fmt` and `tofu validate` as a gate, not as a suggestion
- Write a `tofu test` that fails when the pet length drifts
- Put policy in Open Policy Agent, not in Sentinel
- Scan configuration with Checkov or Trivy before the plan is the first time someone notices

Formatting and Linting: tofu fmt, tofu validate, and Custom Rules

`tofu fmt -recursive` rewrites files to the canonical layout. `tofu fmt -check -recursive` exits non-zero when it would have changed something. The check belongs in the pipeline. The rewrite belongs in a pre-commit hook so the pipeline almost never sees a format failure.

`tofu validate` loads the configuration, checks syntax, and checks that references and types line up. It does not call providers. A directory that has not been initialised can validate only partway. In CI, init with `-backend=false` so validate can see the providers without needing state credentials, then validate.

```
tofu fmt -check -recursive
tofu init -backend=false
tofu validate
```

Custom rules beyond the language are policy, covered below. A linter that reimplements the type system is redundant with validate. A linter that encodes "buckets must have versioning" is policy and should live where policy lives.

OpenTofu 1.13 includes early built-in lint work. Treat it as an extra signal in this series, and keep fmt and validate as the gates you require. Do not fail a production pipeline on an experimental lint until you have read what it considers an error.

Pre-commit Hooks: Catching Errors Before They Reach the Repository

A pre-commit hook runs fmt and validate on the directories the commit touches. Keep the hook fast. Provider downloads inside a hook are how people start skipping the hook.

The hook is a convenience. The pipeline runs the same commands. A skipped hook must not be the only place a check existed. Document the hook as a one-line install in the repository README, and make the CI job the authority.

Useful extra checks in the same hook: reject a commit that adds `terraform.tfstate`, reject `*.tfplan`, and reject a file that looks like a cloud access key. Those are secret checks, not format checks. Chapter 12 lists the patterns.

Testing Strategies: Unit Tests, Integration Tests, and Plan Validation

Three layers, cheapest first.

Static. fmt, validate, policy, and a scanner. No credentials. Every pull request.

Plan tests. `tofu test` with `command = plan` asserts on values the plan can know: names, counts, policy documents built from locals. The `random` provider is enough for the pet module. No cloud account.

Apply tests. `command = apply` creates real objects and destroys them at the end of the test. Run these in an account built for tests, with a budget and a janitor. They do not run on every keystroke. They run on the module when its contract changes, and nightly if you can afford the time.

A plan a reviewer reads is still required for production roots. Tests do not replace that review. They make the review about the infrastructure change rather than about a typo.

OpenTofu Test Framework: Built-in Testing Capabilities

Tests are files named `*.tftest.hcl`. A run block executes plan or apply and asserts.

```
# tests/pet.tftest.hcl
run "two_words" {
  command = plan

  variables {
    pet_length = 2
  }

  assert {
    condition     = random_pet.service.length == 2
    error_message = "The pet resource did not receive length 2."
  }
}
```

`tofu test` from the module directory runs every file. A failing assert fails the command. Variables in the run override defaults. Mocking providers is available when a test would otherwise need a cloud API. Prefer a module boundary you can test with `random` or with pure locals over a large mock that restates the provider.

Test the contract: given this length, the resource is configured with this length, and the output is not null after an apply test. Do not test the provider's implementation. HashiCorp's and the provider's own tests already do that, and a copy of them in your module fails for reasons you cannot fix.

`tofu test` is not a policy engine. An assert that "no security group allows 0.0.0.0/0" is policy that belongs on every repository, not only on modules that remembered to copy the test.

Policy as Code: Sentinel, OPA, and openpolicyagent Integration

Sentinel is HashiCorp's policy language, enforced in their commercial Terraform platform. OpenTofu does not run Sentinel. A `.sentinel` policy pack will not gate `tofu apply`. If a migration guide tells you to point Sentinel at OpenTofu, it is describing the other product.

Open Policy Agent is the policy tool that fits. You evaluate a plan JSON or a configuration JSON with a Rego policy. Conftest is a small runner around that. The usual place is CI, after `tofu plan -out=tfplan` and `tofu show -json tfplan > plan.json`.

```
tofu plan -out=tfplan
tofu show -json tfplan > plan.json
conftest test plan.json --policy policy/
```

A policy that denies `0.0.0.0/0` on a security group, or denies a resource type the team has banned, fails the job before apply. Write Rego against the plan JSON's resource changes, not against a cloud API. The plan is what you are about to do.

Keep policies in a repository the infrastructure repositories consume at a pinned version. A policy copied into each repo will be edited in one of them during an incident and will never return. An exception is a reviewed waiver with an expiry, the same shape as any other break-glass.

Security Scanning: Detecting Misconfigurations with tfsec, checkov, and Trivy

tfsec was a Terraform scanner. Its maintainers folded that work into Trivy. New pipelines should run Trivy or Checkov, not a frozen tfsec binary. The syllabus name is the history. The command you install is one of the living tools.

Both read configuration and look for misconfigurations: public buckets, open security groups, unencrypted disks. They do not replace policy you wrote, and they do not know your naming rules. Run one of them in CI so a well-known mistake fails without a custom Rego rule.

```
trivy config .
```

or:

```
checkov -d . --framework terraform
```

Checkov still says "terraform" for the framework name because the file format is the same. Point it at OpenTofu files. It does not need the Terraform binary.

Triage the first run. A hundred historical findings are not a gate. Turn the gate on for severity you will actually fix, and file the rest. A scanner that fails every build teaches people to ignore scanners. Pin the scanner version the same way you pin OpenTofu, so a rule change is a commit.

Summary

- `tofu fmt -check` and `tofu validate` run in CI after `tofu init -backend=false`
- Pre-commit is the fast path. The pipeline is the authority
- `tofu test` asserts the module contract. Plan tests need no cloud
- Sentinel does not run on OpenTofu. OPA/Conftest evaluates the plan JSON
- Trivy config or Checkov catches known-bad configuration. tfsec's rules live on in that lineage

Security Best Practices and Secrets Management

Security Best Practices and Secrets Management

OpenTofu's privilege is the privilege of the credentials it holds, plus whoever can read the state. This chapter is how to keep both smaller than "the whole cloud."

Learning goals

- Encrypt state in transit and at rest, including client-side encryption
- Authenticate providers with roles, not with long-lived keys in files
- Scope the apply role to the resources this root manages
- Notice the audit record you do and do not get from `tofu` itself

Securing State Files: Encryption at Rest and in Transit

State contains resource attributes. Those include generated passwords if a resource created one, access keys if a resource created one, and the raw values behind anything you marked sensitive. Assume the state is a secret store even when you tried not to put secrets in it.

Three different locks, and you want the ones that match the backend:

Control	What it actually does
TLS to the backend	Stops a network observer reading the state on the wire. The S3, Azure, and GCS backends use HTTPS
Server-side encryption	The bucket or account encrypts bytes at rest. `encrypt = true` on the S3 backend is this. Turn it on
Client-side encryption	OpenTofu encrypts before upload, so the cloud account stores ciphertext. The `encryption` block in chapter 7. `enforced = true` so a misconfiguration cannot write plaintext
Backend IAM	Who may `GetObject` and `PutObject`. This is the real access list
Versioning	Recovers from a bad write. It also retains old secrets until the versions expire. Expire on purpose

A local `terraform.tfstate` on a laptop is an unencrypted secret if the disk is not encrypted, and a stolen secret if the laptop is. Do not leave production state local "just for the migration" longer than that migration's window.

Plans are state-shaped. `tofu plan -out=tfplan` writes sensitive values into the plan file. Delete it after apply. Do not archive plans in the build artifacts without the same encryption as state.

Managing Secrets: Never Hardcoding Credentials

A credential in a `.tf` or `.tfvars` file will be committed, or copied into a plan log, or both. Patterns that hold up:

- Provider credentials from the environment or from a workload identity, never from a `variable` that is stored in tfvars
- Application secrets generated by a secrets manager resource, or written into the secrets manager by its own API, not interpolated into a user-data string that then lands in state in clear text
- `sensitive = true` on the variable so logs redact it. Remember the flag is not encryption
- Ephemeral values and write-only arguments, which OpenTofu 1.11 understands, when the provider offers them. A write-only argument is not stored back into state. Prefer it for a password the provider can accept that way
- A secret that must be passed as a variable comes from `TF_VAR_...` injected by the pipeline for that job, and is not echoed

If a secret has already been committed, rotate it. Deleting the file in a follow-up commit does not remove it from history. Treat the old value as public.

Provider Authentication Patterns: IAM Roles, Service Principals, Workload Identity

The apply process assumes a role that exists only for that run.

AWS. A pipeline's OIDC token assumes an IAM role. No static access key. Locally, a human assumes a role with a short session. The provider block stays `region` only.

Azure. Workload identity federation or a managed identity for the runner. A service principal with a client secret is the legacy path. If you still have the secret, it lives in the pipeline secret store and is injected as environment variables the azurerm provider already understands.

GCP. Workload identity federation from the pipeline, or the attached service account on the runner. `GOOGLE_APPLICATION_CREDENTIALS` pointing at a JSON key on disk is the pattern to retire.

On-premises appliances. Use the provider's token with the same rule: short-lived, injected, scoped to the device the root manages. A shared break-glass password in a provider block is how the network team and the OpenTofu state end up with the same god credential.

Humans and pipelines use different roles. A human role might plan in production and apply only in development. The pipeline role applies in production and cannot be assumed from a laptop. Chapter 13 is where that split sits in the workflow.

Least Privilege for OpenTofu Execution: IAM Policies and RBAC

The apply role can create and destroy whatever the configuration mentions, and anything the provider does as a side effect of those calls. Scope it.

- One role per root module, not one role for the whole company
- Permissions for the resource types that root manages, in the accounts it manages
- A separate, smaller permission set for state: read and write of one bucket prefix, plus the lock, and nothing else. The state role and the infrastructure role can be the same principal in a small team. They should still be two policies you can read
- No permission to edit the state bucket's own policy from the infrastructure role. Otherwise a bad plan can lock you out, or open the state
- Plan in CI can often use the apply role if you accept that plan-time data sources run with apply power. Say so. A read-only plan role is better when data sources allow it, and some providers do not plan cleanly as read-only. Test that before you promise it

Review the role when the configuration gains a new resource type. A pipeline that applies with `AdministratorAccess` will work and will teach nothing about the blast radius.

Audit Logging and Compliance: Tracking Who Changed What and When

OpenTofu writes a plan and a state serial. It is not an audit log. The audit log is the combination of:

Record	Question it answers
The git commit that changed the `.tf`	Who asked for the change, and what the text was
The pipeline run that applied a saved plan	Who approved, when, and which plan digest
Cloud audit logs (CloudTrail, Activity Log, Cloud Audit Logs)	Which API calls the role made
Backend access logs	Who read or wrote the state object
`tofu show` of the plan artifact, if you retain it under the state encryption rules	What the apply was predicted to do

Retain those where your compliance regime says to retain them. A chat message that says "I applied" is not a record. A pipeline that applies from a laptop using a shared user is not a record either.

State encryption does not remove the need for access logs. It changes what a stolen object reveals. Someone who can assume the apply role can still read the decrypted view by running `tofu state pull` with the key. Log that too if the backend and the KMS allow it.

Common Security Pitfalls: Exposed Keys, Overly Permissive Policies, and Insecure Backends

Pitfall	What it looks like	The fix
A key in `terraform.tfvars`	The next commit, or a plan log	Environment or workload identity. Rotate the key that leaked
State in git	`terraform.tfstate` in the repository history	Backend, rotate anything the file held, purge history if the repo is shared
A public state bucket	The backend ACL or a policy of `Principal: *`	Block public access. Review who `GetObject` allows
No lock	Two applies, a corrupt serial	`use_lockfile` or the backend's lock. Refuse to apply without it
`enforced` left off	One init writes plaintext and you do not notice	`enforced = true` after the migration
Apply with an admin role	A bad module creates a second admin	A role per root, reviewed when resources change
A plan file as a build artifact	The artifact store becomes a second state bucket	Encrypt or delete. Do not keep plans next to unrelated logs
Shared `default` workspace	The production apply runs wherever the CLI last sat	Separate roots, or an explicit `workspace select` that fails closed

Summary

- State and plans hold secrets. TLS, server-side encryption, client-side encryption, and IAM are four different controls
- Credentials come from a role the pipeline assumes. They are not file contents
- One apply role per root, plus a tight state policy
- Git, the pipeline, and cloud audit logs together are the audit trail. The CLI is not
- A leaked key is rotated. It is not merely deleted in the next commit

Ephemeral Values and Write-Only Attributes

Ephemeral Values and Write-Only Attributes

Chapter 12 said it plainly: assume state is a secret store. `sensitive = true` hides a value from the terminal, and state encryption protects the file, but the secret is still *in* the file. OpenTofu 1.11 added a third option. An **ephemeral** value exists only for the duration of one `tofu` command and is never written to state or to a saved plan. This chapter covers the whole family on the pinned OpenTofu 1.13.2, and the rules that make it safe.

> **What was and was not run**
>
> The authoring box had no `tofu` binary and the run rules forbid installs, so no block in this chapter was executed. The HCL follows the OpenTofu 1.13 language documentation, the CLI excerpts are copied from it, and the cloud blocks are sketches in the same spirit as the rest of this book. Run `tofu validate` on them before you trust them.

Learning goals

- Tell apart sensitive, encrypted and ephemeral values, and know when you need each
- Use ephemeral variables, ephemeral resources, ephemeral outputs and write-only attributes
- Explain why a write-only attribute needs a companion "version" argument to change
- Recognise the places an ephemeral value may and may not flow, and read the errors

Three different promises

Mechanism	What it promises	Still in state?
`sensitive = true`	The value is redacted in `plan` and `apply` output	Yes, in plaintext unless state is encrypted
State and plan encryption	Whoever reads the stored file without the key sees ciphertext	Yes, encrypted
`ephemeral = true` and ephemeral resources	The value is never written to state or plan at all	No

The OpenTofu documentation recommends using all three together: prefer ephemeral values so the secret is never stored, and if it must be stored, mark it sensitive *and* encrypt state and plans. Ephemeral is not a replacement for the other two. A provider can still echo a value in an error message, and a provisioner's command line is printed even when it contains an ephemeral value, unless you also mark the value sensitive.

Where an ephemeral value may go

An ephemeral value is "infectious": any expression that depends on one is itself ephemeral. OpenTofu then allows it only in these places:

- ephemeral resources
- ephemeral variables
- ephemeral outputs (child modules only)
- locals
- provider configuration blocks
- provisioners and resource `connection` blocks
- write-only attributes of managed resources

Everywhere else (a normal resource argument, `count`, `for_each`, `lifecycle.enabled`, a root-module output) an ephemeral value is a plan-time error. That is the feature working. The error tells you a secret was about to land in state.

Ephemeral input variables

```
variable "db_admin_password" {
  type      = string
  ephemeral = true
}

locals {
  # Depends on an ephemeral variable, so this local is ephemeral too.
  trimmed_password = trimspace(var.db_admin_password)
}
```

OpenTofu does not store the value. Only the variable *name* is kept in a saved plan, so the value must be supplied again when you apply that plan. In practice that means the same `TF_VAR_db_admin_password` environment variable (or `-var`) must be present for both `tofu plan -out=tfplan` and `tofu apply tfplan`. The `tofu apply` documentation has the exact rules for `-var` and `-var-file` with ephemeral root variables. Read it before you wire a pipeline that plans in one job and applies in another.

Ephemeral resources

An `ephemeral` block asks a provider to *open* something for the length of the run and *close* it afterwards: read a secret from a secrets manager, mint a short-lived token, open a tunnel. It is not a drop-in replacement for `data` or `resource`. It works only for types the provider implements as ephemeral, so check the provider's documentation or schema.

```
# Sketch: read an existing secret without storing it.
ephemeral "aws_secretsmanager_secret_version" "db" {
  secret_id = var.db_secret_arn
}

provider "postgresql" {
  host     = var.db_host
  username = "admin"
  password = jsondecode(ephemeral.aws_secretsmanager_secret_version.db.secret_string)["passw
}
```

The lifecycle per run is: validate the configuration with the provider, open the resource to fetch the value, and close it once the current phase no longer needs it. Nothing is recorded, so there is no drift and nothing to import.

The block accepts the usual meta-arguments `depends_on`, `count`, `for_each`, `provider` and `lifecycle`. Inside `lifecycle` only `precondition` and `postcondition` apply, plus `enabled` (with the restriction that `enabled`, like `count` and `for_each`, may not depend on an ephemeral value or on `tofu.applying`).

Deferred opening. If an ephemeral resource's arguments are not fully known during plan (for example the secret ARN comes from a resource that does not exist yet), OpenTofu postpones opening it until apply, and opens it only once its dependencies are satisfied.

Write-only attributes

Reading a secret ephemerally helps only if you can also *hand it* to a resource without that resource storing it. That is a write-only attribute, conventionally suffixed `_wo` by providers:

- it exists only in the configuration of the resource
- it is always written to state and plan as `null`
- the provider always returns it as `null`
- unlike normal arguments, it may reference ephemeral values

The plan renders it as `(write-only attribute)`:

```
  # test_instance.x will be created
  + resource "test_instance" "x" {
      + secret_wo = (write-only attribute)
    }
```

Changing a write-only value

Because state holds `null` and the provider returns `null`, OpenTofu cannot see that the value changed, so it cannot plan an update from the value alone. Providers therefore pair the write-only attribute with a normal attribute that acts as a trigger. The AWS provider's `aws_secretsmanager_secret_version` has `secret_string_wo` and `secret_string_wo_version`:

```
variable "app_secret" {
  type      = map(string)
  ephemeral = true
}

variable "app_secret_version" {
  type    = number
  default = 1
}

resource "aws_secretsmanager_secret_version" "app" {
  secret_id                = aws_secretsmanager_secret.app.id
  secret_string_wo         = jsonencode(var.app_secret)
  secret_string_wo_version = var.app_secret_version # bump to push a new value
}
```

Rotation is now an explicit act: supply the new secret *and* increment `app_secret_version`. If you only change the secret, the plan shows no change. Put that rule in the module README, because it surprises everyone exactly once.

Ephemeral outputs

A child module may return an ephemeral value, but it must say so:

```
# modules/db-credentials/outputs.tf
output "password" {
  value     = jsondecode(ephemeral.aws_secretsmanager_secret_version.db.secret_string)["pass
  ephemeral = true
}
```

Omit `ephemeral = true` and OpenTofu refuses, because the value would otherwise flow into places that are stored. A **root** module output cannot be ephemeral at all. Root outputs are written to state for `terraform_remote_state` consumers, which is exactly what ephemerality forbids.

A complete flow

Put together, a secret can travel from a vault to its consumer without ever touching state:

1. The pipeline injects `TF_VAR_app_secret` (ephemeral variable), or an `ephemeral` block reads it from a secrets manager.
2. Locals reshape it; they become ephemeral automatically.
3. A provider block is configured with it, or a write-only attribute receives it.
4. A child module that needs to hand it on declares an ephemeral output.
5. A `*_wo_version` argument records, in state, only *that* a rotation happened.

The official ephemerality usage example builds this end to end with a module that both stores and reads an AWS Secrets Manager secret.

Diagnosis

Symptom	Likely cause	Fix
Error that an ephemeral value cannot be used in this context	An ephemeral value reached a normal argument, `count`, `for_each` or a root output	Route it through a write-only attribute or provider block, or stop marking it ephemeral if it truly belongs in state
Output error in a child module	The output depends on an ephemeral value but lacks `ephemeral = true`	Add it, and make sure callers use it only in ephemeral contexts
Secret changed, plan shows no changes	Write-only attribute with an unchanged `*_wo_version`	Increment the version argument
`apply tfplan` fails asking for a variable	Ephemeral root variable not supplied again at apply	Export the same `TF_VAR_...` in the apply job
Unknown block type `ephemeral`	Running a release older than 1.11, or the provider has no ephemeral type of that name	Check `tofu version` and the provider schema
Secret visible in provisioner output	Ephemeral does not redact provisioner command lines	Also mark the value `sensitive`

Operational care

- Ephemeral support is per provider and per resource type. Pin provider versions (chapter 6) so a downgrade cannot silently remove a `_wo` attribute.
- Keep state encryption on anyway. Other attributes of the same resources are still stored.
- Audit existing modules for `random_password` and similar resources whose result is stored in state, and for secrets passed through ordinary variables. Those are the migration candidates.
- Remember the companion version argument in reviews: a pull request that changes a secret source without bumping it does nothing.

Practice

1. Take a module that passes a database password through a normal `sensitive` variable to a normal argument. List each place the password is stored today (state, plan files, CI artefacts).
2. Rewrite it with an ephemeral variable and the provider's write-only argument, and add a version variable. Note which stored places disappear.
3. Try to expose the password as a root output and read the error. Then expose it from a child module with `ephemeral = true` and consume it in a provider block.

Recall card

- **Sensitive** hides; **encryption** locks; **ephemeral** never stores.
- Ephemeral flows only into ephemeral resources, variables, child outputs, locals, providers, provisioners, connections and write-only attributes.
- Write-only attributes are `null` in state and need a `*_wo_version` bump to change.
- Root outputs cannot be ephemeral. Ephemeral root variables must be supplied again at apply.

Sources

- OpenTofu, Ephemerality
- OpenTofu, Ephemeral resources
- OpenTofu, Write-only attributes
- OpenTofu, Input variables: ephemerality
- OpenTofu, Output values: ephemeral

Enterprise

CI/CD Integration, GitOps, and Automation Workflows

CI/CD Integration, GitOps, and Automation Workflows

The production habit is: a commit produces a plan, a human approves the plan, and `tofu apply` runs that plan file. This chapter is that pipeline on GitHub Actions, the same shape on other CI systems, and what GitOps does and does not mean for OpenTofu.

Learning goals

- Split plan and apply so the applied plan is the reviewed plan
- Wire a GitHub Actions workflow that uses OpenTofu 1.13.2 and a role, not a key
- Keep approval in front of apply
- Say where Argo CD and Flux fit, and where they do not apply a `.tf` file

The Infrastructure Pipeline: Commit, Plan, Review, Apply

One root module, one pipeline:

1. On a pull request: `tofu fmt -check`, `tofu init -backend=false` or init with read-only state credentials, `tofu validate`, scanner, `tofu plan -out=tfplan`. Comment the plan summary on the request
2. A human reads the plan. Destroy lines and IAM changes are the lines to read first
3. On merge to the branch that is allowed to apply: init with the real backend, plan again, apply the new plan, or apply a plan that was stored and hashed if your change-control rules require the exact bytes from the review
4. The job fails if the workspace, the backend key, or the OpenTofu version is not the one the repository pinned

Re-planning on merge is the common, honest default. The configuration that merges is the configuration you plan. A plan file saved from the pull request goes stale if main moved. If you must apply the exact reviewed bytes, merge by fast-forward only and apply the stored plan before anything else lands. Record the plan digest in the approval.

Never `tofu apply -auto-approve` on a pull request from a fork. Credentials available to untrusted diffs are credentials you have given away. The plan job on a fork can run without credentials when the module's tests allow it, and the apply job runs only on the protected branch.

GitHub Actions Integration: Complete Workflow Examples

Pin the tool. `mise` or the GitHub release action both work. This sketch uses the official setup style and OIDC to AWS. The role trust policy allows this repository's protected environment, not every branch.

```yaml
name: tofu
on:
  pull_request:
  push:
    branches: [main]

permissions:
  id-token: write
  contents: read
  pull-requests: write

jobs:
  plan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: opentofu/setup-opentofu@v1
        with:
          tofu_version: 1.13.2
      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/tofu-plan-pets
          aws-region: eu-west-1
      - run: tofu init
      - run: tofu plan -out=tfplan -no-color
        working-directory: roots/pets
```

Check the current major of `opentofu/setup-opentofu` and `aws-actions/configure-aws-credentials` when you copy this. Actions move. The pins that matter for the tool itself are `tofu_version: 1.13.2` and the commit or tag of the action, which you set to a full SHA in a real repository so a tag cannot move under you.

Apply is a second job, `needs: plan`, `if: github.ref == 'refs/heads/main'`, with the environment `production` requiring reviewers. It runs `tofu apply -auto-approve tfplan` only when `tfplan` was produced in that same workflow by the merge commit. A pull request job stops at plan.

The pet root can use the plan job with no AWS step at all. Keep the cloud role off the toy.

GitLab CI, Jenkins, and Azure DevOps: Multi-Platform Pipeline Patterns

The platform changes. The stages do not.

Stage	GitLab CI	Jenkins	Azure DevOps
Pin	Image `ghcr.io/opentofu/opentofu:1.8.2`	The same image, or `tofu` installed in the agent	The same image
Identity	`id_tokens` and the cloud's OIDC	Workload identity on the agent, not a stored access key	Service connection with federation
Plan	Merge request pipeline	PR build	PR validation
Approve	Protected environment, required approvals	Input step before the apply job	Environment check on the production stage
Apply	Main-branch job	Job after the input, on main	Stage with a condition on the main branch

Write the plan to the job log in a redacted form. `tofu show tfplan` respects sensitive values better than printing state. Store the plan file as an artifact only for the minutes between plan and apply, encrypted at rest by the CI system, and delete it.

A Jenkins agent that holds a static cloud key for every job is the pattern these pipelines exist to leave. One role per root, assumed for that job, shows up in CloudTrail as that role.

GitOps With OpenTofu: Declarative Deployment Using ArgoCD and Flux

GitOps means the git repository is the desired state, and a running process converges the world to it. For Kubernetes, Argo CD and Flux are that process. They apply manifests. They do not natively apply OpenTofu.

Three layouts are honest:

- **Pipeline GitOps.** Git is the source. The CI system in the previous sections is the reconciler. This is the default for cloud accounts and for on-premises appliances. You do not need Argo CD to say you deploy from git
- **OpenTofu provisions the cluster, Flux or Argo CD provisions the workloads.** The cluster's state lives in an OpenTofu root. The workloads live in a git repository those controllers watch. The handoff is an output: a cluster endpoint and a place the controller is installed. Do not also manage the same Deployment with OpenTofu
- **A controller runs OpenTofu.** Projects such as the tofu-controller (the OpenTofu-oriented descendant of Weave's tf-controller) watch a git repository and run plan and apply from inside the cluster. That is GitOps in the strict sense. It belongs when the cluster is already the control plane you operate. It is an extra system to secure: the controller's cloud role, the

source repository it can read, and what it does without a human. Put the approval gate in front of it the same way you would in CI, or accept that merge equals apply and say so

Do not point Argo CD at a directory of `.tf` files and expect a plan. Do not run a second reconciler against a state the pipeline also applies.

Approval Gates and Change Management: Human Review in Automated Workflows

The gate is a protected environment or an explicit approval job between plan and apply. The approver sees the plan summary, the commit, and the OpenTofu version. The approver is not the only author of the change when the repository's rules require two people. For a one-person lab, the gate can be you, and the pipeline still runs the apply so the laptop is not the only place with credentials.

Exceptions during an incident are a break-glass role that is logged and time-limited. The commit still lands. Applying from a laptop with the break-glass role and never pushing the configuration is how state and git diverge. Chapter 10's import and chapter 7's drift are the cleanup if that already happened.

Destructive plans need a louder gate: a second reviewer, or a rule that fails the job unless a label such as `allow-destroy` is present. Encode "this plan destroys" as a check on the plan JSON so it does not depend on someone scrolling.

Parallel Execution and Performance Optimization: Scaling Plans and Applies

OpenTofu walks the graph and applies independent resources in parallel. `-parallelism=n` raises or lowers that fan-out. The default is enough until an API rate-limits you or a provider starts failing under load. Lower it before you raise it. A higher number does not fix a slow data source.

What actually speeds a large root:

- Split roots along the seams in chapter 14 so one plan is not the entire company
- Stop using `depends_on` on whole modules when an output reference is enough
- Keep data sources off moving targets so plans are stable and caches are meaningful
- Run `tofu plan` only in the roots the change touches. A monorepo of many roots does not plan all of them on every commit. Path filters do that
- Init once per job and reuse the provider cache directory when the runner allows it. A cold init of every provider on every plan dominates small configurations

`-refresh=false` skips the refresh and plans against state only. It is faster and it is wrong when the world has drifted. Use it for a local syntax experiment, not for the production apply. There is a narrower "minimal refresh" idea on some Terraform lines. Do not assume the OpenTofu flag set matches Terraform 1.17. Check `tofu plan -help` on 1.13.2.

State locking already serialises applies of one root. Do not also serialise unrelated roots. They have different locks. Parallel pipelines across roots are safe. Parallel pipelines against one state are the lock doing its job: the second waits or fails.

Summary

- Pull requests plan. The protected branch applies the plan it just created
- Pin OpenTofu to 1.13.2 in the workflow. Assume a role. Do not store a cloud key
- GitLab, Jenkins, and Azure DevOps use the same stages
- Argo CD and Flux reconcile Kubernetes. A pipeline or a tofu controller reconciles OpenTofu
- Approval sits between plan and apply. A destroy is a louder approval
- Speed comes from smaller roots and fewer false edges, not from a larger parallelism number

Enterprise-Scale Patterns and Production Operations

Enterprise-Scale Patterns and Production Operations

Enterprise scale is not a bigger state file. It is a set of root modules with owners, a promotion path that does not rely on a workspace name, and an upgrade habit that reads changelogs. This chapter is that layout for cloud and on-premises together.

Learning goals

- Split accounts and on-premises estates into roots with separate state
- Give a team a repository boundary and a shared module, not a shared apply role
- Promote a change from development to production by reviewing the same module at a new pin
- Upgrade OpenTofu and providers as their own commits, with a plan attached

Multi-Cloud and Hybrid Infrastructure: Managing AWS, Azure, GCP, and On-Premises Together

One root module may talk to several providers. A root that talks to all of them because they appear on one architecture slide is a root that fails when any one of them fails, and that only one team is allowed to apply.

Split by blast radius and by credential:

Root	Owns	Does not own
`identity`	The accounts, folders, and projects other roots use	The networks inside them
`network`	Shared networks, once per cloud account	Application buckets
`app-<name>`	That application's resources in one account	The shared network. It takes ids as variables or remote outputs
`onprem-<site>`	Appliances and DNS at that site	The cloud account, except for a narrow handoff such as a VPN endpoint id

Hybrid is two roots and an explicit handoff value, not one configuration with every provider alias. The on-premises provider's token and the cloud role should not sit in the same pipeline job unless the change truly crosses that line in one apply. Most changes do not.

Naming and tags carry the root's identity into the console: `managed-by=opentofu`, the root name, and the repository URL. A human who edits the object can see who will overwrite the edit.

Organizational Structure: Team-Based Repository Layouts and Shared Responsibility

A layout that matches the way reviews work:

```
infra/
  modules/                 # versioned when more than one root calls them
    pets/
  roots/
    pets-dev/
      versions.tf
      backend.tf
      main.tf             # calls modules/pets
    pets-prod/
  policy/                  # Rego, pinned by the pipeline
```

The team that owns the application owns `roots/pets-*` and reviews those plans. A platform team owns `modules/` and the policy repository. They do not need to approve every pet-name change. They do need to approve a new major of a module the application roots pin.

A monorepo is fine when one organisation wants one history. Many repositories are fine when ownership is the point. Do not split so far that a module and its only caller move in lockstep across a version boundary for no reason. Do not stay monolithic so far that a plan for a sandbox prints the production network.

CODEOWNERS, or the equivalent, maps directories to reviewers. The apply role for `pets-prod` is assumable by the production pipeline, not by every developer token. Developers plan in dev. They open a pull request for prod. Chapter 13 is that pipeline.

Environment Strategy: Development, Staging, Production Separation Patterns

Separate roots, separate backend keys, separate roles. Development may use smaller instances and may allow destroy. Production sets `prevent_destroy` on the data you cannot recreate and requires the approval gate.

Promotion is "the prod root moves its module pin from 1.4.2 to 1.5.0", not "we apply the dev workspace with a different variable." The module version is the artifact. The root's variables are

the environment. If prod and dev call different code, you are not promoting. You are maintaining two systems.

Staging exists when a change can break a dependency you cannot see in dev, such as a provider behaviour against a real load balancer. It is another root, smaller than production, with production-like configuration. It is not a workspace name inside the production state.

Ephemeral environments, one per pull request, are a root created by automation with a backend key that includes the request id, destroyed when the request closes. They need a budget, a janitor for leftovers, and a ban on using production data. They are a later optimisation, not the first layout.

Versioning and Upgrades: OpenTofu Versions, Provider Updates, and Breaking Changes

Keep a matrix and look at it on a schedule. As of 21 October 2026 the supported OpenTofu lines are 1.13 (until 1 August 2027) and 1.12 (until 1 February 2027). 1.11 and older are out of security support. This book uses 1.13.2. A production estate on 1.6 is not "stable." It is unpatched.

Upgrade OpenTofu in its own change:

1. Read the series changelog, including the breaking notes. For 1.13 those include WinRM provisioner removal, `base64gzip` byte changes, macOS 13, and the end of 32-bit builds
2. Bump the pin in mise or `.terraform-version` and `required_version`
3. `tofu init` and `tofu plan` in a non-production root
4. Read the plan. An empty plan is the goal. A surprise replace is a stop
5. Apply non-production, then production, with the normal gate

Provider upgrades are a different change, even when you do them the same week. `-upgrade` rewrites the lock file. Commit that diff alone. Major provider versions get a changelog reading and a plan in dev before prod's pin moves.

Modules follow semver. A root that uses `~> 1.4` will take 1.4.z on the next init without a commit if you also forgot the lock. Commit the lock. Take minors on purpose.

Do not upgrade OpenTofu, the AWS provider, and the network module in one pull request. A bad plan then has three causes.

Debugging and Logging: Tracing Failures in Complex Deployments

`tofu plan` and the error it prints are the first tool. Provider errors include the API message. Read it before you turn on tracing.

`TF_LOG=info` is enough for "which resource, which retry." `TF_LOG=trace` is enormous and includes values that may be sensitive. Use trace on a throwaway reproduction, write it to a file you delete, and do not paste it into a ticket without reading it.

`TF_LOG_PROVIDER=debug` on one provider keeps the core quiet. `TF_LOG_PATH` writes the log to a file instead of mixing it with the plan.

When a plan hangs, it is usually a lock, a provider waiting on a credential, or a data source waiting on an API. Check the lock before you kill the process. Killing an apply can leave the lock and can leave an object half-created. The next plan tells you which. Resume from a clean init rather than from a half-written local `.terraform`.

Crash logs (`crash.log`) go with the bug report to the provider or to OpenTofu, not into the repository. Reproduce on 1.13.2 and the provider version from the lock file, or the report is incomplete.

Real-World Case Studies: Production Patterns from Leading Organizations

Three patterns cover most estates this book is willing to call production. They are patterns, not claims about a particular company's internals.

The single-application team. One repository, two roots (`dev` and `prod`), one module for the application, GitHub Actions with OIDC, state in a locked and encrypted bucket the application role cannot reconfigure. Plans on pull requests. Apply on main after one approval. This is the pet example grown up, and it is enough for a product company that is not a cloud provider.

The platform team and many application roots. The platform team publishes modules and Rego. Application teams pin them. Each application has its own state and its own role. The platform team does not apply application changes. A weekly job reports roots whose OpenTofu pin is not 1.13.x. The report is a pull request, not a silent upgrade.

The hybrid estate. Cloud roots as above. An on-premises root per site, with the appliance provider pinned and the token injected per job. The only shared values are outputs: an IP, a BGP peer, a DNS zone id. A change that touches both sides is two pull requests with an order written on them, applied in that order, each with an empty follow-up plan.

What all three refuse: a single state for every environment, a shared admin key, an apply from a laptop as the normal path, and an upgrade with no plan attached.

Summary

- Split roots by credential and blast radius, including on-premises sites
- Teams own roots. A platform team owns shared modules and policy
- Dev, staging, and prod are separate states. Promotion moves a module pin
- Upgrade OpenTofu and providers in separate commits, on a supported series
- Log at info first. Trace is a reproduction tool, and it leaks values
- A production pattern has a locked backend, a role per root, and a plan in the pipeline

Conclusion: Delivering on the Infrastructure-as-Code Promise

Conclusion: Delivering on the Infrastructure-as-Code Promise

Infrastructure as code is delivered when a second person can apply the same change you would have applied, from the commit you reviewed, and get an empty plan the next time. The binary being OpenTofu matters for the license, the pin, and a few features Terraform does not have. It does not change that test.

What you can do this quarter

Start from the pet module and grow one real root. In order:

1. Pin OpenTofu 1.13.2. Commit the lock file and `.gitignore` before the first real resource
2. Put state in a backend with a lock, versioning, server-side encryption, and the client-side `encryption` block once you have somewhere to keep the key
3. Split variables, outputs, and provider configuration so a review has somewhere to look
4. Add `tofu fmt`, `tofu validate`, and either Trivy or Checkov to the pull request. Add one Rego rule you actually care about
5. Plan on the pull request. Apply on the protected branch with a role the laptop does not have
6. Import existing objects one group at a time, and stop when the plan is empty

A team that does those six things is operating OpenTofu. Workspaces, a controller inside Kubernetes, and a module published to the registry can wait until a second environment or a second caller makes them the smaller design.

What the promise does not include

The configuration is not the whole system. Consoles, controllers, and people with admin roles write to the same APIs. The plan is how you notice. Ignoring drift, or setting `ignore_changes = all`, gives the promise back.

Sentinel, HCP Terraform, and Stacks are not waiting inside the `tofu` binary. Policy is OPA. The control plane is your pipeline. Composition is a root module that calls children.

OpenTofu 1.13 and Terraform 1.17 are different tools that share an ancestor. Pin one per state. Read the changelog when the pin moves. The 1.13 series is supported until 1 August 2027, and support is not a permanent condition.

From first principles to production

You define the end state in HCL. Providers make it true. State remembers the ids. Modules give that definition a boundary other people can call. Tests and policy keep the boundary honest. The pipeline is the only routine way the state is allowed to change.

That is the whole arc. The next apply should be boring.