Terraform / Infrastructure as Code

A Complete 11-Chapter Infrastructure Provisioning Course

Topics covered:

Why Terraform exists & the HCL language · Variables, outputs & locals

State management & remote backends · Modules & the Terraform Registry

Workspaces & multi-environment patterns · Provisioners, import & drift

CI/CD & testing · Capstone: a real multi-resource environment

Exercises: 33 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · builds directly on the site's own Cloud Platforms course

Table of Contents

CHAPTER 1 OF 11

What Terraform Is & Why It Exists

Terraform / Infrastructure as Code

Chapter 1 · What Terraform Is & Why It Exists

`cloud1-11` gave Infrastructure as Code a first look — declarative vs. imperative, Terraform named as the cross-provider option, plan/apply and configuration drift introduced at a conceptual level. This course goes deep specifically on Terraform itself: not IaC in general, but this one tool, in real depth.

The Configuration Drift Problem

A server gets provisioned by hand through a cloud console. Six months later, nobody fully remembers which settings were changed, when, or why — a firewall rule added during an incident, a instance size bumped up "temporarily" during a traffic spike and never reverted. Staging quietly stops matching production. This is **configuration drift**, and it's the exact problem `cloud1-11` named but didn't solve: infrastructure that exists only in a console, with no record of its own history, and no reliable way to reproduce it.

Declarative vs. Imperative Infrastructure

An **imperative** script says exactly what steps to run, in order: "create a VM, then attach a disk, then open port 443." Run it twice and it may fail the second time (the VM already exists) or duplicate resources. A **declarative** tool like Terraform instead says what the end state should look like — "there should be one VM, with this disk, with this port open" — and lets Terraform figure out what actions are needed to get there, including doing nothing if reality already matches.

```
# A conceptual preview — real syntax and installation come in Chapter 2
resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t3.micro"
}
```

That block doesn't say "run these commands" — it says "this instance should exist, with these properties." Terraform compares that declared intent against what's actually running and works out the difference itself.

Terraform vs. the Cloud-Native Alternatives

TOOL	PROVIDER SCOPE	LANGUAGE
Terraform	Cross-provider — AWS, Azure, GCP, and hundreds of others via a shared provider model	HCL (HashiCorp Configuration Language)
CloudFormation	AWS only	JSON/YAML
ARM / Bicep	Azure only	JSON / Bicep DSL
Deployment Manager	GCP only	YAML/Jinja2/Python

Each cloud-native tool is genuinely well-integrated with its own provider — but locked to it. Terraform's core value, especially relevant given `cloud1-2`'s own cross-provider terminology map, is expressing infrastructure across providers (or a multi-cloud environment) using one consistent language and workflow.

Terraform vs. Ansible — Two Different Jobs

Terraform **provisions** infrastructure — it creates the VM, the network, the database instance. It has no concept of "log in and install a package." Ansible **configures** infrastructure that already exists — installing software, editing config files, restarting services, on machines Terraform (or something else) has already brought into being. They're complementary, not competing: a real pipeline commonly runs Terraform first to create the servers, then Ansible to configure them.

The Terraform Workflow: Write → Plan → Apply → Destroy

- **Write** — describe desired infrastructure in `.tf` configuration files
- **Plan** — Terraform compares desired state against actual state and shows exactly what it intends to change, before touching anything
- **Apply** — Terraform executes that plan, creating/updating/deleting only what's needed
- **Destroy** — Terraform tears down everything it manages, cleanly, in dependency order

The `plan` step is the single biggest practical difference from clicking through a console: you see the change before it happens, every time.

When Terraform Fits — and When It Doesn't

A one-off resource for quick local testing, or a genuinely single, tiny, rarely-touched environment, often isn't worth the ceremony. Terraform earns its keep once infrastructure needs to be reproducible, reviewed by a

team, or exist in more than one environment (dev/staging/prod) — exactly the scenarios where console-driven drift becomes a real, recurring problem.

THE PLAN STEP IS THE POINT

Most of Terraform's real-world value comes from one habit: never apply a change you haven't first reviewed via `plan`. It's the single practice most responsible for catching unintended changes before they happen.

PROVISIONING ≠ CONFIGURING

Don't reach for Terraform to install software or edit files on a running server — that's Ansible's job (still on this site's own bucket list). Mixing the two responsibilities into one tool is a common early mistake.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why a declarative tool comparing desired state against actual state avoids the "run it twice and it breaks" problem that an imperative script has.

View solution

EXERCISE 2

A team wants to manage infrastructure spread across AWS and Azure with one consistent workflow. Explain why CloudFormation is a poor fit here, and why Terraform is.

View solution

EXERCISE 3

A colleague suggests using Terraform to install and configure nginx on an existing server. Explain why this is the wrong tool for that specific job, and name the right one.

View solution

CHAPTER 1 QUICK REFERENCE

- **Configuration drift** — infrastructure changed by hand, with no record, that quietly diverges from what's expected
- **Declarative** — describe the desired end state; the tool figures out the steps
- **Terraform vs. CloudFormation/ARM/Deployment Manager** — cross-provider vs. single-cloud-native

- **Terraform vs. Ansible** — provisions infrastructure vs. configures infrastructure that already exists
- **Workflow** — write → plan → apply → destroy, with `plan` as the review step before anything changes

CHAPTER 2 OF 11

Installing Terraform & Your First Configuration

Terraform / Infrastructure as Code

Chapter 2 · Installing Terraform & Your First Configuration

Chapter 1's `resource` block was a preview. This chapter installs the real CLI and runs an actual configuration end to end — deliberately using a provider that needs no cloud account or credentials, so the workflow itself is the entire focus.

Installing Terraform

Terraform ships as a single binary — download it from HashiCorp's releases, or install via a package manager (`brew install terraform` on macOS, `choco install terraform` on Windows, or an apt/dnf repository on Linux). Verify the install:

```
$ terraform version
Terraform v1.9.0
```

The `terraform` and `provider` Blocks

Every configuration starts with a `terraform` block declaring which providers it needs, and a `provider` block configuring one of them. A **provider** is a plugin that knows how to talk to a specific system — AWS, Azure, a local filesystem, even things like GitHub or Cloudflare all have providers.

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.5"
    }
  }
}

provider "local" {}
```

`terraform init`

`init` reads the `required_providers` block, downloads the matching provider plugin(s) into a local `.terraform/` directory, and initializes the backend (by default, a plain local state file — remote backends are Chapter 6's own topic).

```
$ terraform init

Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/local versions matching "~> 2.5"...
- Installing hashicorp/local v2.5.1...

Terraform has been successfully initialized!
```

The Dependency Lock File

`init` also writes `.terraform.lock.hcl`, recording the exact provider version and checksum actually installed. This is the same idea as `node1-2`'s `package-lock.json` or `ruby2-7`'s `Gemfile.lock`: it guarantees everyone running `init` against this configuration gets the identical provider build, not just "something matching the version constraint."

- **Commit `.terraform.lock.hcl` to git** — it's what makes provider versions reproducible across machines and teammates
- **Never commit `.terraform/`** — it's a local download cache, regenerated by `init` on any machine, exactly like `node_modules/` or a Rust `target/` directory

Provider Version Pinning

The `version = "~> 2.5"` constraint means "any 2.5.x release, but not 2.6.0 or higher." Pinning matters for the same reason the lock file does — an un-pinned provider could silently pick up a newer version with different behavior, producing configuration drift of its own, this time in the tool itself rather than the infrastructure it manages.

The `resource` Block In Depth

A resource has the shape `resource "<type>" "<local name>" { ... }`. The type (`local_file`) is defined by the provider; the local name (`hello`) is how this configuration refers to it. Together they form a unique **resource address**, `local_file.hello`, used throughout the rest of the configuration and in CLI commands.

```
resource "local_file" "hello" {
  content  = "Hello from Terraform!"
  filename = "${path.module}/hello.txt"
}
```

terraform plan

Run `plan` before ever applying — it shows exactly what will change, using `+` for create, `-` for destroy, and `~` for update-in-place.

```
$ terraform plan

Terraform will perform the following actions:

  # local_file.hello will be created
  + resource "local_file" "hello" {
      + content  = "Hello from Terraform!"
      + filename = "./hello.txt"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

terraform apply

`apply` re-runs the same plan and, after an interactive confirmation, executes it.

```
$ terraform apply
...
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

local_file.hello: Creating...
local_file.hello: Creation complete after 0s

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

`hello.txt` now exists on disk. Run `terraform plan` again with nothing changed, and it reports **"No changes"** — the same idempotency property from Chapter 1's Exercise 1, now demonstrated for real.

`terraform destroy`

Tears down everything this configuration manages, in dependency order, after its own interactive confirmation.

```
$ terraform destroy
...
local_file.hello: Destroying...
local_file.hello: Destruction complete after 0s

Destroy complete! Resources: 0 added, 0 changed, 1 destroyed.
```

A SAFE FIRST PROVIDER

The `local` provider needs no account, no credentials, and no cost — deliberately chosen here so the workflow itself (init → plan → apply → destroy) is what gets practiced first, before real cloud providers enter the picture.

NEVER HARDCODE CREDENTIALS IN A PROVIDER BLOCK

A real cloud provider block (e.g. `provider "aws" {}`) should read credentials from environment variables or a credentials file — never write an access key directly into a `.tf` file. Exactly the rule `pipelines1-5` and `crypto1-11` already established: a committed credential is compromised the moment it's pushed, permanently, even if later removed.

Hands-On Exercises

EXERCISE 1

After applying the `hello.txt` example, running `terraform plan` again with nothing changed prints "No changes." Explain what this output confirms about the workflow, connecting it back to Chapter 1's idempotency exercise.

View solution

EXERCISE 2

Explain why `.terraform.lock.hcl` should be committed to git while the `.terraform/` directory should not, drawing the parallel to another ecosystem's own lock-file convention.

View solution

EXERCISE 3

A colleague writes `provider "aws" { access_key = "AKIA..." secret_key = "..." }` directly in a committed `.tf` file. Explain why this is dangerous and what to do instead.

View solution

CHAPTER 2 QUICK REFERENCE

- `terraform init` — downloads providers, initializes the backend, writes the lock file
- `terraform plan` — shows what would change (`+` / `-` / `~`), changes nothing
- `terraform apply` — executes the plan after confirmation
- `terraform destroy` — tears down every managed resource
- **Commit** `.terraform.lock.hcl` — **never commit** `.terraform/`
- Never hardcode credentials in a provider block — use environment variables or a credentials file

CHAPTER 3 OF 11

The Terraform Language In Depth

Terraform / Infrastructure as Code

Chapter 3 · The Terraform Language In Depth

Chapter 2 wrote one resource in isolation. Real infrastructure is many resources referencing each other — this chapter covers how HCL expresses those relationships, and the meta-arguments that control how resources are created.

HCL Syntax Building Blocks

A block has a type, zero or more labels, and a body of arguments: `resource "type" "name" { key = value }`. Values can be strings, numbers, booleans, lists (`["a", "b"]`), or maps (`{ key = "value" }`). String interpolation embeds an expression inside a string with `${...}`, and comments use `#` or `/* */`.

```
variable "env" { default = "dev" }

resource "local_file" "config" {
  # interpolation combines a literal string and an expression
  filename = "${path.module}/${var.env}-config.txt"
}
```

Data Sources

A `resource` block creates and manages something. A `data` block only **reads** something that already exists — infrastructure this configuration doesn't own, like a shared VPC created elsewhere, or the latest AMI ID published by a cloud provider.

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]
}

resource "aws_instance" "web" {
  ami = data.aws_ami.ubuntu.id
}
```

The Implicit Dependency Graph

`data.aws_ami.ubuntu.id` above is a reference — and referencing one resource's attribute from inside another is exactly what tells Terraform "create/read this one first." Terraform builds this into a full dependency graph (a DAG) automatically, from every reference in the configuration, and applies resources in the correct order — running independent resources in parallel where nothing connects them. **File order never matters** — only the actual references do.

`count`

Creates multiple copies of a resource from a single block, indexed numerically via `count.index`.

```
resource "local_file" "server_config" {
  count    = 3
  filename = "${path.module}/server-${count.index}.txt"
}
```

Resource addresses become `local_file.server_config[0]`, `[1]`, `[2]`.

`for_each`

Creates multiple copies keyed by a map or set of strings, via `each.key` / `each.value` — the modern default for anything beyond a fixed, never-changing count.

```
resource "local_file" "server_config" {
  for_each = toset(["web", "api", "worker"])
  filename = "${path.module}/${each.key}.txt"
}
```

Resource addresses become `local_file.server_config["web"]`, `["api"]`, `["worker"]` — keyed by a stable name, not a position.

> **COUNT VS. FOR_EACH — REMOVING FROM THE MIDDLE**
>
> With `count = 3`, removing the resource at index 1 shifts index 2 down to fill the gap — Terraform sees this as "destroy index 1 AND index 2, then recreate a new index 1," not "destroy the one item that was actually removed." With `for_each`, deleting `"api"` from the set only ever affects the resource keyed `"api"` — the other two are untouched, because their addresses were never based on position in the first place.

`depends_on`

For the rare case where one resource genuinely depends on another but no attribute reference exists to express it (e.g. IAM permission propagation delays), `depends_on` states the dependency explicitly. Used sparingly — an implicit reference is almost always preferable when one is available, since it stays correct even if the configuration is later refactored.

```
resource "aws_instance" "web" {
  depends_on = [aws_iam_role_policy.web_policy]
}
```

The `lifecycle` Block

A meta-argument block controlling special-case behavior for how a resource is created, updated, or destroyed:

```
resource "aws_instance" "web" {
  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [tags]
  }
}
```

- `create_before_destroy` — builds the replacement before tearing down the original, avoiding downtime on a forced replacement
- `prevent_destroy` — makes `terraform destroy` (or a plan that would destroy this resource) fail outright, a safety rail for genuinely critical resources
- `ignore_changes` — tells Terraform to stop reporting drift on specific attributes, e.g. tags a separate system updates automatically

DEFAULT TO FOR_EACH

Unless a collection is guaranteed to always be exactly N items with no individual identity, `for_each`'s stable, name-based addressing avoids the reordering trap `count` can cause.

Hands-On Exercises

EXERCISE 1

Explain why referencing one resource's attribute inside another resource's argument is what creates Terraform's dependency graph, and why the order the blocks appear in the file doesn't matter.

View solution

EXERCISE 2

A team manages 3 identical resources with `count = 3` and needs to delete only the middle one (index 1). Explain what Terraform plans to do to the other two resources, and why `for_each` avoids this problem.

View solution

EXERCISE 3

Explain what `prevent_destroy` protects against, and describe a concrete real-world resource where it's genuinely worth setting.

View solution

CHAPTER 3 QUICK REFERENCE

- **data** — reads existing infrastructure; never creates or manages it
- **Dependency graph** — built from references (`resource.attribute`), not file order
- **count** — numeric index, position-based addresses, reordering risk
- **for_each** — key-based addresses, stable under add/remove — the modern default
- **depends_on** — explicit dependency, used only when no implicit reference exists
- **lifecycle** — `create_before_destroy`, `prevent_destroy`, `ignore_changes`

CHAPTER 4 OF 11

Variables, Outputs & Locals

Terraform / Infrastructure as Code

Chapter 4 · Variables, Outputs & Locals

Every value so far has been hardcoded directly into a resource block. Real configurations need to be parameterized — the same configuration deployed to dev, staging, and prod, with only a handful of values actually differing between them.

Input Variables

A `variable` block declares a named input, with an optional type, default, and description. Reference it anywhere in the configuration with `var.<name>`.

```
variable "instance_count" {
  type        = number
  default     = 2
  description = "How many instances to create"
}

resource "local_file" "config" {
  for_each = toset(range(var.instance_count))
  filename = "${path.module}/instance-${each.key}.txt"
}
```

Omitting `default` makes the variable **required** — Terraform refuses to plan until a value is supplied, useful for forcing an explicit choice (like an environment name) rather than silently falling back to a guess.

Variable Validation

A `validation` block rejects bad input at plan time, before anything is ever created — cheaper than discovering the mistake mid-apply.

```
variable "environment" {
  type = string
  validation {
    condition     = contains(["dev", "staging", "prod"], var.environment)
    error_message = "environment must be dev, staging, or prod."
  }
}
```

Sensitive Variables

Marking a variable `sensitive = true` redacts its value from CLI plan/apply output — Terraform prints `(sensitive value)` instead of the real string.

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

> **SENSITIVE HIDES OUTPUT, NOT THE STATE FILE**
>
> `sensitive = true` only redacts CLI output — the real value is still written into the state file in plaintext. Protecting that file is Chapter 5's own topic; marking a variable sensitive is not, by itself, a complete solution to keeping a secret safe.

Locals

A `locals` block defines internal computed values — not inputs, just names for expressions used more than once. Unlike variables, locals aren't set from outside the configuration; they exist purely to avoid repeating the same expression.

```
locals {
  name_prefix = "${var.environment}-${var.project}"
}

resource "local_file" "config" {
  filename = "${path.module}/${local.name_prefix}-config.txt"
}
```

`.tfvars` Files

Instead of typing `-var` flags on every run, values can live in a `.tfvars` file. `terraform.tfvars` and any `*.auto.tfvars` file load automatically; anything else needs an explicit `-var-file` flag — the pattern Chapter 8's workspaces build on for genuinely separate `dev.tfvars` / `prod.tfvars` files per environment.

```
# terraform.tfvars
environment     = "dev"
instance_count  = 2
```

Variable Precedence Order

Lowest to highest priority — later sources override earlier ones:

1. Environment variables (`TF_VAR_name`)
2. `terraform.tfvars`, if present
3. `terraform.tfvars.json`, if present
4. Any `*.auto.tfvars` / `*.auto.tfvars.json` files, in alphabetical order
5. `-var` and `-var-file` command-line flags, in the order given — **always win**

CLI FLAGS ALWAYS WIN

A `-var` flag on the command line overrides every file-based source — useful for one-off overrides during testing, but easy to forget about when a plan doesn't match what the `.tfvars` file seems to say.

Hands-On Exercises

EXERCISE 1

Explain why `sensitive = true` alone is not sufficient to protect a secret value, and name what else actually needs protecting.

View solution

EXERCISE 2

`terraform.tfvars` sets `region = "us-east-1"`, and the run also passes `-var="region=us-west-2"` on the command line. Which value does Terraform actually use, and why?

View solution

EXERCISE 3

Explain the practical difference between a variable and a local, and describe a scenario where each is the right choice.

View solution

CHAPTER 4 QUICK REFERENCE

- **variable** — an input; omit `default` to make it required
- **validation** — rejects bad input at plan time
- **sensitive = true** — redacts CLI output only, not the state file
- **locals** — internal computed values, not external inputs
- **Precedence (low → high)** — env vars → `terraform.tfvars` → `*.auto.tfvars` (alphabetical) → `-var` / `-var-file` flags

CHAPTER 5 OF 11

State Management

Terraform / Infrastructure as Code

Chapter 5 · State Management

Chapter 4's warn-box left a thread open: state stores real secret values in plaintext. This chapter explains what state actually is, why Terraform can't function without it, and what that means for how it must be handled.

What State Is & Why Terraform Needs It

Every `apply` writes `terraform.tfstate` — a JSON file mapping every resource address (`local_file.hello`) to the real-world object it created and every attribute that object actually has. Without it, Terraform would have no way to distinguish "a resource this configuration already created" from "a resource with the same name that happens to exist for some other reason" — every `plan` would risk creating duplicates rather than recognizing something as already satisfied. State is the literal record that makes the desired-vs-actual comparison from Chapter 1 possible at all.

Local State

By default, state lives as a plain file on local disk — fine for solo experimentation, but it means only one machine has any record of what Terraform manages. Anyone else running `apply` against the same configuration, from a different machine, has no shared state to compare against.

NEVER COMMIT TERRAFORM.TFSTATE TO GIT

State contains the full, real attribute values of every managed resource — including anything marked `sensitive` in Chapter 4, stored here in plain, unredacted text. Committing it is exactly the same mistake as Chapter 2's hardcoded-credential warning: permanently exposed in git history the moment it's pushed. Add `*.tfstate*` to `.gitignore`, the same discipline `git1-6` already covers.

Remote State — A Preview

Local state has two problems at once: secrets sitting in a file that's tempting to accidentally commit, and no shared record for a team. Chapter 6 covers **remote backends** in full — storing state in a shared, access-controlled location instead of on one person's disk, solving both problems together.

`terraform state` Commands

Inspecting and safely modifying state without hand-editing the file:

```
$ terraform state list
local_file.hello

$ terraform state show local_file.hello
# local_file.hello:
resource "local_file" "hello" {
    content  = "Hello from Terraform!"
    filename = "./hello.txt"
}

$ terraform state mv local_file.hello local_file.greeting
$ terraform state rm local_file.old_resource
```

`state mv` renames a resource in state without destroying and recreating the real object — Terraform otherwise treats a renamed block exactly like Chapter 3's addressing rules imply: a different address means a different resource, and a different resource means destroy-then-create.

> **NEVER HAND-EDIT THE STATE FILE**
>
> State's internal structure is easy to corrupt by editing directly — a single malformed field can make Terraform lose track of a resource entirely. Use `terraform state mv` / `rm` / `list` / `show` instead; they modify state safely, through Terraform's own understanding of its structure.

Drift Detection, In Full

`cloud1-11` introduced drift conceptually. Here's the actual mechanism: `terraform plan` performs a three-way comparison — the **desired configuration** (the `.tf` files), the **last-known state** (what Terraform believes exists), and a fresh **refresh** against the real infrastructure itself. If a manual console change altered something Terraform manages, the refresh step detects that the real world no longer matches state, and `plan` reports it — exactly the "manual fix gets silently reverted by the next automated apply" scenario `cloud1-11` warned about.

SOURCE	WHAT IT REPRESENTS
Configuration	What the `.tf` files say should exist
State	What Terraform last recorded as existing
Real infrastructure	What's actually running right now, checked via refresh

Automatic Backups

Every state-modifying command writes `terraform.tfstate.backup` before making its change — a single-generation safety net if a state operation goes wrong.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Terraform needs a state file at all — what specifically would break if it tried to operate without one?

View solution

EXERCISE 2

Explain the three-way comparison `terraform plan` performs, and how it's able to detect a manual console change as drift.

View solution

EXERCISE 3

A teammate wants to rename a resource block from `"web"` to `"app_server"` without Terraform destroying and recreating the real infrastructure. What command should they use, and why is hand-editing the state file instead a real risk?

View solution

CHAPTER 5 QUICK REFERENCE

- **State** — the record mapping configuration to real-world resources; required for desired-vs-actual comparison
- **Never commit** `*.tfstate*` — plaintext secrets, same risk as a hardcoded credential
- `terraform state list` / `show` / `mv` / `rm` — safe ways to inspect and modify state
- **Never hand-edit** the state file directly
- **Drift** — detected by comparing configuration, state, and a real-infrastructure refresh, three ways

CHAPTER 6 OF 11

Remote Backends & Team Collaboration

Terraform / Infrastructure as Code

Chapter 6 · Remote Backends & Team Collaboration

Chapter 5 flagged two problems with local state: secrets sitting in a file tempting to accidentally commit, and no shared record for a team. Remote backends solve both.

Why Local State Breaks With a Team

A second teammate cloning the repository has zero record of what the first person's `apply` already created — their local `terraform.tfstate` simply doesn't exist yet. Worse, if two people *do* somehow share a state file and both run `apply` at nearly the same time, each reads the same starting state, computes a plan against it, and both try to write an updated state back afterward — whichever write happens last silently overwrites the other, potentially losing track of resources the first apply just created.

Remote Backends

A `backend` block inside `terraform {}` moves state storage off local disk entirely, into a shared, access-controlled location every team member and CI runner reads from and writes to.

```
terraform {
  backend "s3" {
    bucket         = "my-company-tfstate"
    key            = "projects/web-app/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

The S3 + DynamoDB Backend

The classic AWS pattern: an S3 bucket holds the state file itself (with `encrypt = true` for encryption at rest — the same principle `dbsec1-5` / `crypto1-5/6` already covered), and a DynamoDB table provides locking. Every

teammate and CI job reads/writes the exact same object in S3, so there's genuinely one shared source of truth instead of N different local copies.

State Locking

Before writing to state, Terraform first acquires a lock — a conditional write to the DynamoDB table that only succeeds if no lock currently exists. If someone else's `apply` already holds the lock, Terraform blocks with a clear error rather than racing to write state at the same time:

```
Error: Error acquiring the state lock

Lock Info:
  ID:        7c6f...
  Path:      projects/web-app/terraform.tfstate
  Operation: OperationTypeApply
  Who:       teammate@ci-runner-3
```

This is exactly what prevents the concurrent-write corruption described above — only one `apply` can ever be mid-flight against a given piece of state at a time.

Terraform Cloud / HCP Terraform

A managed alternative to hand-rolling S3+DynamoDB — built-in state storage, locking, a web UI, and run history, with no separate bucket or lock table to provision yourself. Genuinely the "batteries included" option, at the cost of depending on a hosted service rather than infrastructure the team fully owns.

Backend Migration

Switching from local state to a remote backend (or between two remote backends) isn't automatic — it requires an explicit migration:

```
$ terraform init -migrate-state
Initializing the backend...
Terraform detected that the backend type changed...
Do you want to copy existing state to the new backend?
  Enter a value: yes
```

After migrating, every teammate needs to re-run `terraform init` themselves — their local Terraform still points at the old backend configuration until they do.

THIS IS WHAT ACTUALLY PROTECTS THE SECRETS IN STATE

Chapter 4 and Chapter 5 both flagged that a secret's plaintext lives in state regardless of the `sensitive` flag. An encrypted S3 bucket with IAM access restricted to only the people/systems that genuinely need it is the real answer to that open question — protecting the storage location itself, since the file's contents can't be hidden from anyone who has already been granted read access.

FORCE-UNLOCK IS A LAST RESORT

A crashed CI job can leave a lock stuck indefinitely, blocking every future apply. `terraform force-unlock <lock-id>` exists for exactly this — but confirm no apply is genuinely still in progress first. Force-unlocking while a real apply is actively running risks the exact concurrent-write corruption locking exists to prevent.

Hands-On Exercises

EXERCISE 1

Explain concretely what can go wrong if two team members run `terraform apply` at nearly the same time against shared local state with no locking mechanism.

View solution

EXERCISE 2

In the S3 + DynamoDB backend pattern, explain DynamoDB's specific role, and what could still go wrong even with state safely stored remotely in S3 if DynamoDB locking weren't part of the setup.

View solution

EXERCISE 3

A CI job crashes mid-apply and leaves a stuck lock, blocking the team. Explain why `terraform force-unlock` should be used cautiously, and what should be verified before running it.

View solution

CHAPTER 6 QUICK REFERENCE

- **Remote backend** — moves state to a shared, access-controlled location; solves both the secrets-in-git risk and team collaboration
- **S3 + DynamoDB** — S3 stores state (encrypted at rest), DynamoDB provides locking

- **Lock** — only one `apply` can hold it at a time, preventing concurrent state writes
- **Terraform Cloud/HCP** — managed alternative with built-in storage, locking, and a web UI
- `terraform init -migrate-state` — required for any backend change; every teammate must re-init afterward
- `terraform force-unlock` — last resort only, after confirming no apply is genuinely still running

CHAPTER 7 OF 11

Modules

Terraform / Infrastructure as Code

Chapter 7 · Modules

Everything so far has lived in a single, flat set of files. Real infrastructure repeats the same shapes — a web server, a VPC, a database tier — over and over, across environments and projects. Modules package a shape once and reuse it.

What a Module Is

Any directory of `.tf` files is a module. Whichever one `terraform` commands are run from is the **root module** — every configuration in this course so far has technically already been one, just never explicitly calling any others.

Child Modules

A `module` block calls another module from the root (or from another module), passing inputs as arguments matching that module's own declared variables, and reading its declared outputs back.

```
module "web_server" {
  source        = "./modules/web-server"
  instance_type = "t3.micro"
  environment   = var.environment
}

output "web_ip" {
  value = module.web_server.public_ip
}
```

Module Structure & Composition

A module's own directory follows the same `variables.tf` / `main.tf` / `outputs.tf` convention as any configuration — inputs come in through its `variable` blocks, results go out through its `output` blocks. Everything else inside stays private to the module; the caller never sees its internal resource names.

```
# modules/web-server/variables.tf
variable "instance_type" { type = string }
variable "environment"   { type = string }

# modules/web-server/outputs.tf
output "public_ip" {
  value = aws_instance.this.public_ip
}
```

The Public Terraform Registry

`registry.terraform.io` hosts community and vendor-maintained modules — a well-tested VPC module, for instance, covers subnet layout, route tables, NAT gateways, and dozens of edge cases a first attempt at writing one from scratch would likely miss.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"

  cidr = "10.0.0.0/16"
}
```

Module Versioning

Registry modules use the same `~>` version-constraint syntax as providers (Chapter 2) — pinning a module version protects against an upstream update introducing a breaking change without warning, exactly the same reproducibility guarantee the provider lock file already provides.

DRY Infrastructure

Module blocks accept `for_each` and `count` too — calling the same module multiple times with different inputs, rather than copy-pasting a resource shape once per environment.

```
module "web_server" {
  for_each      = toset(["dev", "staging"])
  source        = "./modules/web-server"
  environment   = each.key
}
```

A MODULE'S VARIABLES.TF IS ITS API

Anyone calling a module only interacts with its declared inputs and outputs — treat that contract with the same care as a public function signature. Changing a required variable's meaning breaks every caller, exactly like a breaking API change would.

DON'T MODULARIZE A SINGLE RESOURCE WITH NO REAL LOGIC

A module wrapping one plain resource, with no meaningful composition or reuse behind it, adds a layer of indirection without earning it back — matching the general principle that an abstraction should be introduced because the task genuinely needs it, not by default. Reach for a module once a shape is actually reused, or genuinely complex enough to be worth naming.

Hands-On Exercises

EXERCISE 1

Explain the difference between the root module and a child module, and explain in what sense every Terraform configuration in this course so far has technically already "been" a module.

View solution

EXERCISE 2

A team needs a VPC matching a very common, well-established pattern. Explain why using a vetted community module from the Terraform Registry is often preferable to writing the VPC resources from scratch, and what version pinning specifically protects against.

View solution

EXERCISE 3

A colleague wraps a single resource block, with no meaningful added logic, in its own module. Explain why this might be premature abstraction, and what tradeoff is actually being made.

View solution

CHAPTER 7 QUICK REFERENCE

- **Root module** — where commands are run from; **child module** — called via a `module` block
- Every configuration is technically a module — the root module, specifically
- **Registry** — `registry.terraform.io`, vetted community/vendor modules

- Module `version` — same `~>` syntax as providers, protects against breaking upstream changes
- `for_each` / `count` on a `module` block — DRY, reused across environments
- Don't modularize a single resource with no real logic behind it

CHAPTER 8 OF 11

Workspaces & Multi-Environment Patterns

Terraform / Infrastructure as Code

Chapter 8 · Workspaces & Multi-Environment Patterns

Chapter 4's `.tfvars` files and Chapter 7's `for_each`-over-a-module both hinted at multiple environments. This chapter names the real options directly, and the tradeoffs between them.

The Multi-Environment Problem

Dev, staging, and prod are almost always the same infrastructure shape with different values — smaller instance sizes in dev, different domain names, different scaling targets. The question is how to run one configuration against three genuinely separate sets of real infrastructure without copy-pasting the whole thing three times.

Terraform Workspaces

A workspace gives each named environment its own state file, within the *same* backend and configuration. `terraform.workspace` is available inside the configuration itself to vary behavior by environment.

```
$ terraform workspace new dev
$ terraform workspace new prod
$ terraform workspace select dev
$ terraform workspace show
dev
```

```
resource "local_file" "config" {
  count    = terraform.workspace == "prod" ? 3 : 1
  filename = "${path.module}/${terraform.workspace}-${count.index}.txt"
}
```

Workspaces vs. Separate State Keys

An alternative to workspace commands entirely: give each environment its own backend key (Chapter 6's S3 `key` argument) via `-backend-config`, with no `terraform workspace` involved. Functionally similar —

separate state per environment — but explicit at the backend-configuration level rather than an in-CLI selection that's easy to forget you've made.

Workspaces vs. Separate Directories

The strongest-isolation option: a genuinely separate directory (and root module) per environment — `environments/dev/`, `environments/prod/` — each with its own `main.tf` calling the same shared modules from Chapter 7's registry pattern, but with entirely separate state, separate backend configuration, and often separate provider credentials.

APPROACH	ISOLATION	DUPLICATION
Workspaces	Separate state only — same config, backend, credentials	Lowest — one configuration
Separate state keys	Separate state, explicit backend config per environment	Low — one configuration, per-env backend files
Separate directories	Strongest — state, backend, and often credentials all separate	Highest — a root module per environment

The Workspace-Isn't-a-Full-Isolation-Boundary Gotcha

Workspaces separate **state** — nothing else. The configuration, the backend, and the provider credentials are all shared across every workspace. Selecting the wrong workspace by mistake — intending `dev` but currently sitting in `prod` — still runs `apply` against real production infrastructure, using real production credentials, because nothing in the setup itself enforces which workspace "should" be safe to touch right now.

A NAMING CONVENTION, NOT A SECURITY BOUNDARY

Because credentials and configuration are shared, many teams deliberately choose separate directories (or at least separate backend configs with separate credentials) for genuinely high-stakes environments like production — specifically because workspaces don't provide a hard barrier against running the wrong command against the wrong environment.

CHECK BEFORE EVERY APPLY

`terraform workspace show` before running `apply` is a cheap habit against exactly this mistake — easy to forget which workspace is currently selected, especially after switching between projects.

Hands-On Exercises

EXERCISE 1

Explain precisely what `terraform.workspace` isolates, and what it does *not* isolate — and why that makes running `apply` in the wrong workspace still genuinely dangerous.

View solution

EXERCISE 2

Compare workspaces against separate directories in terms of isolation strength and duplication cost. Which would a team managing a genuinely high-stakes production environment likely prefer, and why?

View solution

EXERCISE 3

Given `count = terraform.workspace == "prod" ? 3 : 1`, a team creates a third workspace called `staging`. Explain exactly what `count` evaluates to while the `staging` workspace is selected, and why.

View solution

CHAPTER 8 QUICK REFERENCE

- `terraform workspace new` / `select` / `list` / `show` — manage and check the current workspace
- `terraform.workspace` — the current workspace name, usable inside configuration
- **Workspaces isolate state only** — configuration, backend, and credentials are shared
- **Separate directories** — strongest isolation, highest duplication, often preferred for prod
- Always check `terraform workspace show` before `apply`

CHAPTER 9 OF 11

Provisioners, Import & Handling Drift

Terraform / Infrastructure as Code

Chapter 9 · Provisioners, Import & Handling Drift

Every prior chapter assumed clean, fully-managed resources. Real infrastructure has messier edges — a one-time bootstrap step with no native equivalent, or a resource that already existed before Terraform ever touched it. This chapter covers the deliberate escape hatches for exactly those cases.

Provisioners — `local-exec` and `remote-exec`

A `provisioner` block runs a command as a side effect of a resource's creation — `local-exec` on the machine running Terraform itself, `remote-exec` on the resource that was just created, via SSH or WinRM.

```
resource "aws_instance" "web" {
  provisioner "remote-exec" {
    inline = ["sudo systemctl start nginx"]
  }
}
```

Why Provisioners Are Discouraged

A provisioner's command is **imperative code embedded inside a declarative resource** — the exact split Chapter 1 opened with. Terraform has no visibility into what that script actually did, can't record its effect in state, and can't detect or reconcile drift on it the way it can for a real resource attribute. HashiCorp's own guidance is to treat provisioners as a last resort — and per Chapter 1's Terraform-vs-Ansible distinction, using `remote-exec` to configure a server is arguably doing Ansible's job from inside Terraform, the exact mixing of responsibilities Chapter 1's warn-box already cautioned against.

When Provisioners Are Still Justified

A genuinely one-time bootstrap action with no native resource or provider API equivalent, or triggering an external signal/webhook right after creation. Not a substitute for real configuration management — if the goal is installing and maintaining software on a server, that's still Ansible's job.

PROVISIONERS RUN ONCE, ON CREATION, BY DEFAULT

A provisioner fires when its resource is first created — not on every subsequent `apply`. Editing a provisioner's command doesn't re-run it against a resource that already exists; only creating a new one (or an explicit `when = destroy` provisioner, or a `null_resource` with `triggers`) causes it to run again.

Importing Existing Infrastructure

`terraform import` brings a resource created outside Terraform — by hand in a console, or by another tool — under Terraform's management, without destroying and recreating it.

```
$ terraform import aws_instance.web i-0abcdef1234567890
```

IMPORT ONLY CREATES THE STATE ENTRY

`import` populates state — it does *not* write the matching `resource` block for you. The resource block still has to be written by hand first, as a best guess at the real object's actual configuration, or the next `plan` will show Terraform trying to "fix" every attribute it thinks is wrong.

The Import Workflow in Practice

1. Write a `resource` block matching the real object as closely as possible
2. Run `terraform import <address> <real-world-id>`
3. Run `terraform plan` — any remaining mismatch shows up as a planned change
4. Adjust the resource block, repeat step 3, until `plan` reports no changes

Newer Terraform versions (1.5+) support a declarative `import` block plus config generation, letting Terraform draft the resource block for you — a materially easier modern alternative to hand-writing it from scratch, though the underlying iterate-until-`plan`-shows-nothing workflow is unchanged.

PLAN IS THE VERIFICATION STEP, EVERY TIME

Immediately after any import, run `plan`. It's the only reliable signal that the hand-written resource block genuinely matches reality — the same "plan before you trust anything" habit from Chapter 1, applied here to a freshly-imported resource specifically.

Resolving `cloud1-11`'s Own Gotcha

`cloud1-11` warned that a manual console fix gets silently reverted by the next automated apply. Now the real fix: if a manual change genuinely needs to persist, either **(a)** update the `.tf` configuration itself to match the new reality, so the next `plan` shows no diff, or **(b)** if the resource was created entirely outside Terraform, bring it under management with `import` first. Re-running `apply` repeatedly and hoping the drift stops happening is never a real fix — Terraform will keep reverting toward whatever the configuration says, every single time, until the configuration itself changes.

Hands-On Exercises

EXERCISE 1

Explain why embedding a `local-exec` / `remote-exec` provisioner breaks Terraform's own desired-state model from Chapter 1 — specifically, what can Terraform no longer track about the provisioner's effect?

View solution

EXERCISE 2

A resource was created by hand in the AWS console before this project's Terraform configuration existed. Describe the two-step workflow needed to bring it under Terraform management safely, including the common surprise about what `import` alone does and doesn't do.

View solution

EXERCISE 3

A manual console fix was applied during an incident to keep a service running. Explain the two legitimate options for stopping the automated apply from reverting it, and why "just don't run apply for a while" isn't a real fix.

View solution

CHAPTER 9 QUICK REFERENCE

- **Provisioners** — imperative escape hatch, discouraged, last-resort only
- Provisioners run once, on creation, by default — not on every apply
- `terraform import` — brings an existing resource under management without recreating it
- Import only writes state — the resource block must still be written by hand (or via a config-generation import block on 1.5+)
- Always `plan` immediately after an import to verify the resource block truly matches reality

- Fixing drift for real: update configuration to match, or import — never just keep re-applying

CHAPTER 10 OF 11

Terraform in CI/CD & Testing

Terraform / Infrastructure as Code

Chapter 10 · Terraform in CI/CD & Testing

Every command so far has been run by hand. A real team runs Terraform through CI — this chapter ties the whole workflow together using `pipelines1`'s own material, applied specifically to infrastructure changes.

`terraform fmt` & `validate`

`fmt` rewrites files into Terraform's canonical style; `fmt -check` exits non-zero without changing anything, suitable for a CI gate. `validate` catches syntax and type errors — without touching real infrastructure or needing full provider credentials, just an initialized backend.

```
$ terraform fmt -check
$ terraform validate
Success! The configuration is valid.
```

The Plan-as-PR-Comment Pattern

A CI job runs `terraform plan` on every pull request and posts the output as a comment directly on that PR — reviewers see *exactly* what would change in real infrastructure before ever approving the merge, not just the raw `.tf` diff. A one-line variable change can produce a plan that recreates a dozen resources; the diff alone would never show that.

A Real GitHub Actions Workflow

Following `pipelines1-3`'s workflow-YAML pattern and `pipelines1-6`'s job-dependency gating:

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

jobs:
  plan:
    if: github.event_name == 'pull_request'
    steps:
      - run: terraform fmt -check
      - run: terraform validate
      - run: terraform plan -out=tfplan
      - # post tfplan output as a PR comment

  apply:
    if: github.event_name == 'push'
    needs: []
    steps:
      - run: terraform apply -auto-approve
```

Policy as Code

Automated rules that block a plan from ever being applied if it violates a defined policy — "no public S3 buckets," "no unencrypted volumes," enforced mechanically rather than relying on a reviewer noticing. **Sentinel** is HashiCorp's own policy engine (Terraform Cloud/Enterprise); **OPA** (via `conftest`) is the open-source alternative usable in any CI pipeline. Both directly enforce the security practices `dbsec1` / `crypto1` / `owasp1` already covered — the difference is these rules run automatically on every plan, not just when a reviewer happens to remember to check.

`terraform test` & Terratest

Terraform's native testing framework (1.6+) uses `.tftest.hcl` files with `run` blocks and `assert` conditions, and can execute against mocked providers — fast, free tests with no real infrastructure spun up. **Terratest** (a third-party, Go-based library, older and still widely used) takes the opposite approach: it actually provisions real infrastructure to test against, then tears it down — slower and billable, but validates genuinely real provider behavior rather than a mock's approximation of it.

Bringing It Together — the Full CI/CD Gate

`fmt` / `validate` → `test` → `plan` + policy check → human review of the posted plan comment → merge → `apply`. The same test → build → staging → production chain `pipelines1-9` 's own capstone assembled for

application code, applied here to infrastructure changes instead.

AUTOMATION CATCHES SYNTAX; A HUMAN STILL CATCHES INTENT

Even with every automated gate in place, the human reading the posted `plan` output remains the single most valuable checkpoint — automation reliably catches format/policy/syntax violations, but "wait, that's not actually what I meant to change" is still a human judgment call.

APPLY ONLY FROM THE REVIEWED, MERGED BRANCH — AND MIND WHO HOLDS THE CREDENTIALS

Never let CI apply directly from a feature branch or on every push — only from `main`, after review, mirroring `pipelines1-8`'s protected-environment gating. Worth stating honestly: this CI-driven model is **push-based** — the CI runner itself needs real deployment credentials, the same tradeoff `k8s2-9` named for traditional CI/CD. Tools like Atlantis or Terraform Cloud's own run system move toward a more pull-based model, but the GitHub Actions pattern shown here is push-based by default.

Hands-On Exercises

EXERCISE 1

Explain why `terraform fmt -check` and `validate` run before `plan` in a CI pipeline rather than after, and what each one catches that the other doesn't.

View solution

EXERCISE 2

Explain the value of the plan-as-PR-comment pattern — what does it let a human reviewer see that reading the raw `.tf` file diff alone wouldn't show?

View solution

EXERCISE 3

Contrast policy as code (Sentinel/OPA) enforcement against a manual code-review checklist for catching something like "no public S3 buckets." Why can automated enforcement catch cases a checklist-based review might miss?

View solution

CHAPTER 10 QUICK REFERENCE

- `terraform fmt -check` — style; `validate` — syntax/type errors, no real infra touched
- **Plan-as-PR-comment** — shows the real infrastructure impact before merge, not just the file diff
- **Policy as code** — Sentinel (HashiCorp/Enterprise) or OPA/conftest (open), automated policy enforcement on every plan
- `terraform test` — native, mockable, fast; **Terratest** — real infrastructure, slower, more realistic
- Apply only from the reviewed, merged branch — this CI model is push-based, so the runner holds real credentials

CHAPTER 11 OF 11

Capstone: Provisioning a Real Multi-Resource Environment

Terraform / Infrastructure as Code

Chapter 11 · Capstone — Provisioning a Real Multi-Resource Environment

Ten chapters, one piece at a time. This closing chapter combines every one of them into a single, realistic three-tier environment — a VPC, a compute tier, and a database — built and deployed the way a real team actually would.

The Target Environment

A VPC with public and private subnets, a web tier running in the public subnets, and a database in the private subnets — a genuinely common shape, deliberately chosen so every chapter's own technique has a real place to be applied.

Step 1 — Remote Backend & Locking

Configured first, before anything else exists: Chapter 6's S3+DynamoDB backend, so state is safely shared and locked from the very first `apply` onward — never starting on local state and migrating later.

```
terraform {
  backend "s3" {
    bucket         = "company-tfstate"
    key            = "capstone/terraform.tfstate"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

Step 2 — The VPC via a Registry Module

Chapter 7: the VPC itself is a vetted community module, not hand-written — its subnet layout, route tables, and NAT gateways already cover edge cases a first attempt wouldn't.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"
  cidr    = "10.0.0.0/16"
}
```

Step 3 — Variables & Environment-Specific Values

Chapter 4: `instance_count`, an `environment` variable with validation, and a `sensitive` `db_password` — with Chapter 5's own honest caveat still true: `sensitive` hides CLI output, the encrypted backend from Step 1 is what actually protects the value at rest.

Step 4 — The Compute Tier With `for_each`

Chapter 3: web servers created with `for_each` over a stable set of names, referencing the VPC module's own subnet-ID outputs — the dependency graph resolving automatically, exactly as Chapter 3 first demonstrated with a data source.

```
resource "aws_instance" "web" {
  for_each  = toset(["a", "b"])
  subnet_id = module.vpc.public_subnets[0]
}
```

Step 5 — The Database Tier With `lifecycle` Protection

Chapter 3's own Exercise 3 scenario, applied for real: the database gets `prevent_destroy = true` — a deliberate safeguard against exactly the accidental-destroy risk that exercise walked through.

```
resource "aws_db_instance" "main" {
  lifecycle {
    prevent_destroy = true
  }
}
```

Step 6 — Environments: Separate Directories, Not Workspaces

Chapter 8 concluded that workspaces alone aren't a strong enough boundary for a genuinely high-stakes production environment — this capstone deliberately follows that conclusion, using separate

`environments/dev/` and `environments/prod/` directories, each with its own backend key and credentials, rather than `terraform workspace select`.

Step 7 — Handling a Pre-Existing Resource

The DNS record for this environment was created by hand, before this project existed. Chapter 9's two-step import workflow brings it under management: write the matching `resource` block, run `terraform import`, then iterate against `plan` until it reports no changes.

Step 8 — The CI/CD Gate

Chapter 10's full pipeline wraps the whole thing: `fmt`/`validate` → `terraform test` → `plan` posted as a PR comment, gated by a policy-as-code check → human review → merge → `apply`, running only from the reviewed `main` branch.

The Full Picture

STEP	CHAPTER
1	Ch.6 — Remote backend & locking
2	Ch.7 — VPC via a registry module
3	Ch.4 — Variables, validation, sensitive values
4	Ch.3 — `for_each` and the dependency graph
5	Ch.3 — `lifecycle` protection
6	Ch.8 — Environment isolation strategy
7	Ch.9 — Import
8	Ch.10 — CI/CD gate

What's Still Out of Scope, Honestly

This capstone provisions infrastructure — it deliberately does not configure servers or install software, exactly the Terraform-vs-Ansible boundary Chapter 1 drew from the very first page; that's still a genuinely separate, still-outstanding piece of this site's own bucket list. Multi-region disaster recovery, cost optimization at real scale (`cloud1-9`/`cloud2-7`), and a dedicated secrets manager beyond an encrypted backend are all real next steps a production system would eventually need, named honestly rather than treated as solved here.

THE THROUGH-LINE, RESTATED

Every step in this capstone is a direct, concrete application of a chapter's own material — nothing new was introduced here. That's deliberate: production-readiness comes from combining well-understood, individually simple pieces correctly, not from any single advanced trick.

TERRAFORM PROVISIONS; IT DOESN'T CONFIGURE

Nothing in this capstone installs or configures software on the web tier once it exists — that boundary, drawn all the way back in Chapter 1, holds even at capstone scale. A real deployment still needs Ansible (or an equivalent) for that half of the job.

Closing the Course

From Chapter 1's configuration-drift problem to a real, CI-gated, multi-resource environment — every chapter in between solved one specific, genuine piece of that original problem. State, locking, modules, environments, import, and automated review together are what actually make infrastructure reproducible, reviewable, and safe to change — not any single command on its own.

Hands-On Exercises

EXERCISE 1

Explain why Step 1 (the remote backend) had to be configured before any other resource in this capstone, rather than being added later once the environment already existed.

View solution

EXERCISE 2

Explain why this capstone chose separate directories over Terraform workspaces for dev/prod isolation, tying your answer back to the specific reasoning Chapter 8 gave.

View solution

EXERCISE 3

Across the entire course, name the single idea that recurs most often, and explain in your own words why understanding it matters more than memorizing any individual command.

View solution

CHAPTER 11 QUICK REFERENCE — COURSE COMPLETE

- Every capstone step is a direct, unmodified application of an earlier chapter's own technique
- Still out of scope: server configuration (Ansible's job), DR at scale, dedicated secrets management
- Terraform provisions infrastructure; it never configures what runs on top of it
- **Course complete** — Terraform / Infrastructure as Code, 11 chapters