

# **Terraform A Visual Guide**

Foundations, Maintainable Code, and Safe Operations

Reiner Braun

Sample Book — Chapters 1 to 2

# Introduction

This book is designed for readers who are taking their first steps with Terraform, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

## How to Use This Book

**Read the chapters in order on your first pass.** The sequence builds the required background step by step: the operating model comes before commands, commands before stateful team workflows, and team workflows before imports, refactoring, automation, and recovery.

At the beginning of each chapter, review **What You'll Be Able to Do** before reading the detailed sections. These outcomes identify the practical understanding the chapter is designed to develop and help you distinguish core knowledge from supporting detail.

Use the diagrams to understand **relationships, processes, boundaries, and structures** that are harder to reconstruct from prose alone. A diagram supports the explanation; it does not replace the prerequisites, limitations, or operational cautions in the surrounding text.

Try the code and hands-on examples yourself whenever possible. Use a disposable local directory or an approved non-production environment, confirm the stated prerequisites, read the plan before applying it, and verify the result rather than assuming that command completion proves success.

At the end of each chapter, use the **Chapter Summary** to restate the main ideas in your own words, then answer the **Review Questions** without looking back. Revisit the relevant section when an answer is incomplete or uncertain.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as **signposts**: understand why the topic exists, where it fits, and when you may need to study it in greater depth during practical work.

# Table of Contents

## [Introduction](#)

## [How to Use This Book](#)

## [Part I — Build the Correct Foundation](#)

### [Chapter 1 — Terraform in One Mental Model](#)

[What You'll Be Able to Do](#)

[The central idea](#)

[The four parts of Terraform's operating model](#)

[Configuration: the desired result](#)

[Terraform Core: the coordinator](#)

[Providers: the API specialists](#)

[State: the ownership record](#)

[A blueprint analogy](#)

[Write, plan, apply](#)

[Write](#)

[Plan](#)

[Apply](#)

[Terraform follows dependencies, not file order](#)

[Desired configuration, recorded state, and actual infrastructure](#)

[What Terraform manages well](#)

[Terraform manages declared ownership, not an entire account](#)

[The practical value](#)

[Three misconceptions to remove immediately](#)

[Chapter Summary](#)

[Review Questions](#)

### [Chapter 2 — A Safe First Terraform Run](#)

[What You'll Be Able to Do](#)

[Complete the full lifecycle before touching shared infrastructure](#)

[Verify the executable](#)

[Create an isolated working directory](#)

[Initialize the directory](#)

[Format and validate before planning](#)

[Preview the first change](#)

[Apply the configuration](#)

[Make one controlled change](#)

[Clean up deliberately](#)

[Know which generated files matter](#)

[Generated files are not interchangeable](#)

[What this first run proves](#)

[Chapter Summary](#)

[Review Questions](#)

## **Part I — Build the Correct Foundation**

# Chapter 1 — Terraform in One Mental Model

## What You'll Be Able to Do

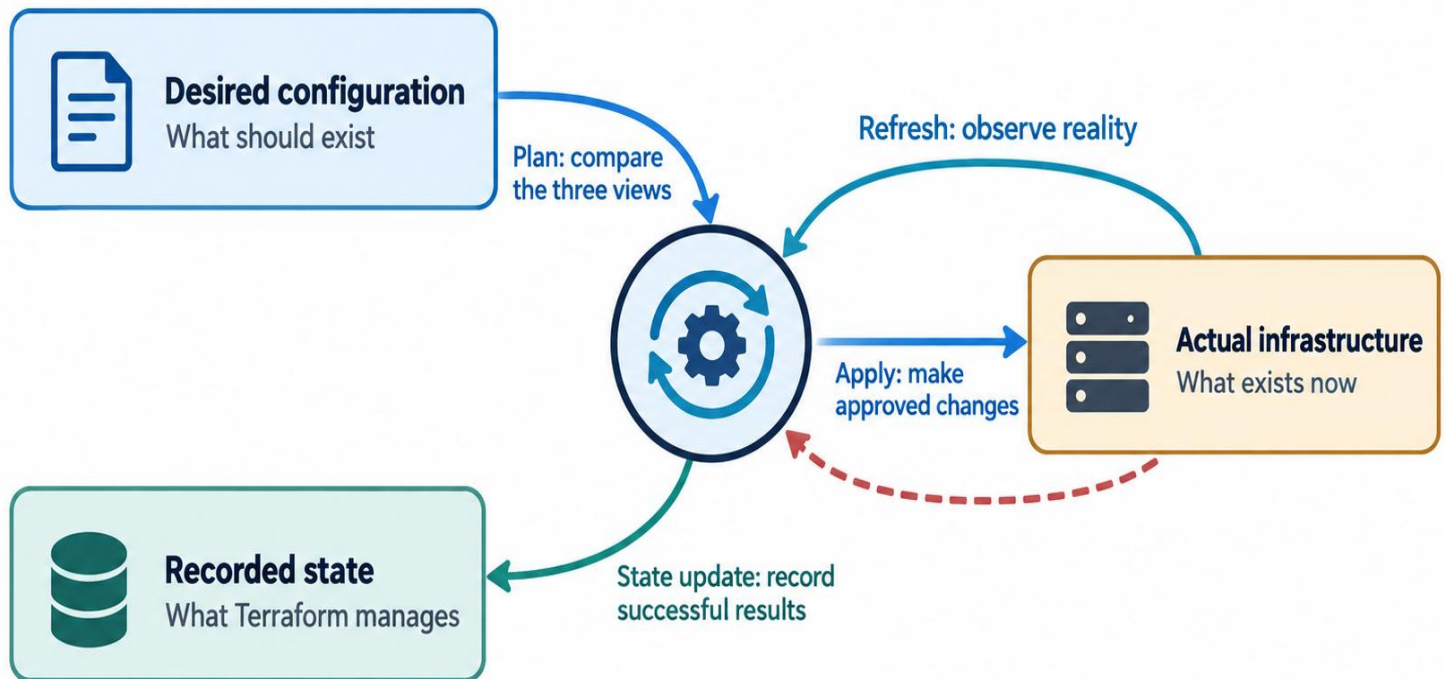
- Explain Terraform as a **stateful reconciliation system** rather than a top-to-bottom script.
- Distinguish the roles of configuration, Terraform Core, providers, state, plans, and remote systems.
- Explain why references and dependencies—not file order—normally determine operation order.
- Recognize which visible objects Terraform owns and which objects it merely reads.

## The central idea

Terraform repeatedly reconciles three views of infrastructure: what the configuration says should exist, what the state says Terraform manages, and what the target systems report actually exists. Most beginner errors come from treating one of these views as though it were the whole system.

# Chapter 1: Terraform in One Mental Model

## 1-1 Three-View Reconciliation



## Three views, one reconciliation loop

001

Figure 1-1: Three-View Reconciliation

**Infrastructure as code**, often shortened to **IaC**, is the practice of defining and changing infrastructure through versioned, reviewable code rather than relying only on manual console actions. IaC does not make infrastructure risk-free. It makes intent visible, repeatable, and easier to review.

Terraform uses a **declarative** model. A declarative configuration states the desired result, while an imperative script states the sequence of steps to perform. The difference is similar to ordering a finished meal rather than dictating every movement the cook must make.

An imperative script might say:

1. Call an API to create a network.

2. Read the returned network identifier.
3. Call another API to create a subnet in that network.
4. Call a third API to create a server in that subnet.

A Terraform configuration instead describes a network, a subnet that refers to the network, and a server that refers to the subnet. Terraform derives the dependency order from those references.

## The four parts of Terraform's operating model

### Configuration: the desired result

A **Terraform configuration** is a collection of `.tf` files that declare infrastructure objects and the values connecting them. The configuration answers questions such as:

- Which objects should Terraform manage?
- What properties should those objects have?
- Which values come from inputs?
- Which objects depend on values produced by other objects?

Configuration is the team's statement of intent. It should be stored in version control and reviewed like application code.

### Terraform Core: the coordinator

**Terraform Core** is the main Terraform program. It reads configuration, evaluates expressions, constructs a dependency graph, asks providers to read or change objects, compares values, and writes updated state.

Terraform Core does not contain built-in knowledge of every cloud or SaaS API. It knows how Terraform configuration, dependency evaluation, planning, and state management work.

### Providers: the API specialists

A **provider** is a plugin that teaches Terraform how to work with a particular API or system. Examples include providers for public clouds, Kubernetes, source-control platforms, monitoring services, and local files.

A provider defines the resource types and data sources available for its system. It also converts Terraform's requests into API operations and converts API responses back into values Terraform can use.

### State: the ownership record

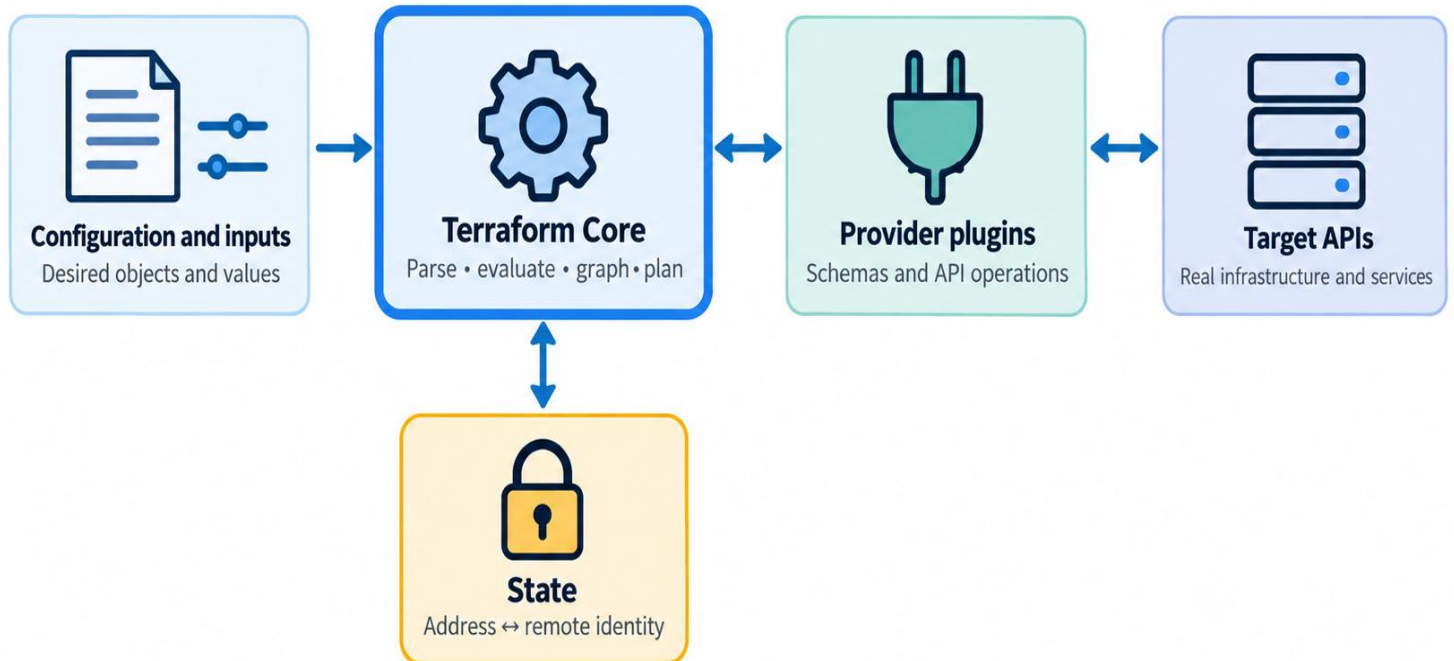
**State** is Terraform's persisted record of managed resource instances and their corresponding remote object identities. State answers a question that configuration alone cannot answer: "Which real object is represented by this resource address?"

For example, configuration might declare `aws_vpc.application`, but only state records that this address is associated with a particular VPC identifier such as `vpc-0123456789abcdef0`.

State also stores attributes needed for future comparisons. Because those attributes can include sensitive values, **state** must be protected as sensitive operational data.

# Chapter 1: Terraform in One Mental Model

## 1-2 Responsibilities in Terraform's Operating Model



### Core coordinates; providers specialize

002

Figure 1-2: Responsibilities in Terraform's Operating Model

### A blueprint analogy

Think of a construction project:

- The **configuration** is the approved blueprint.
- The **state** is the project register that maps each blueprint item to a real installed object.
- The **providers** are specialist contractors who know how to work with electrical, plumbing, or structural systems.
- The **plan** is a change proposal showing what would be built, modified, or removed.
- The **apply** is the authorized construction work.

The blueprint does not prove that the building currently matches it. The project register does not replace the building. The contractor can encounter unavailable materials or rejected permits. In the same way, configuration, state, providers, and real infrastructure have different roles.

### Write, plan, apply

Terraform's core workflow has three stages.

#### Write

You create or modify configuration that expresses the intended infrastructure. At this stage, normal engineering practices matter: clear names, version control, focused changes, and peer review.

## Plan

A **plan** is Terraform's proposed set of actions. Terraform reads the configuration and state, asks providers about relevant remote objects, and determines whether each managed object should be created, updated, replaced, read, or destroyed.

A **plan is a proposal, not a completed change**. Reviewing only the final count—such as “2 to add, 1 to change, 0 to destroy”—is not enough. A reviewer must inspect which objects and attributes will change.

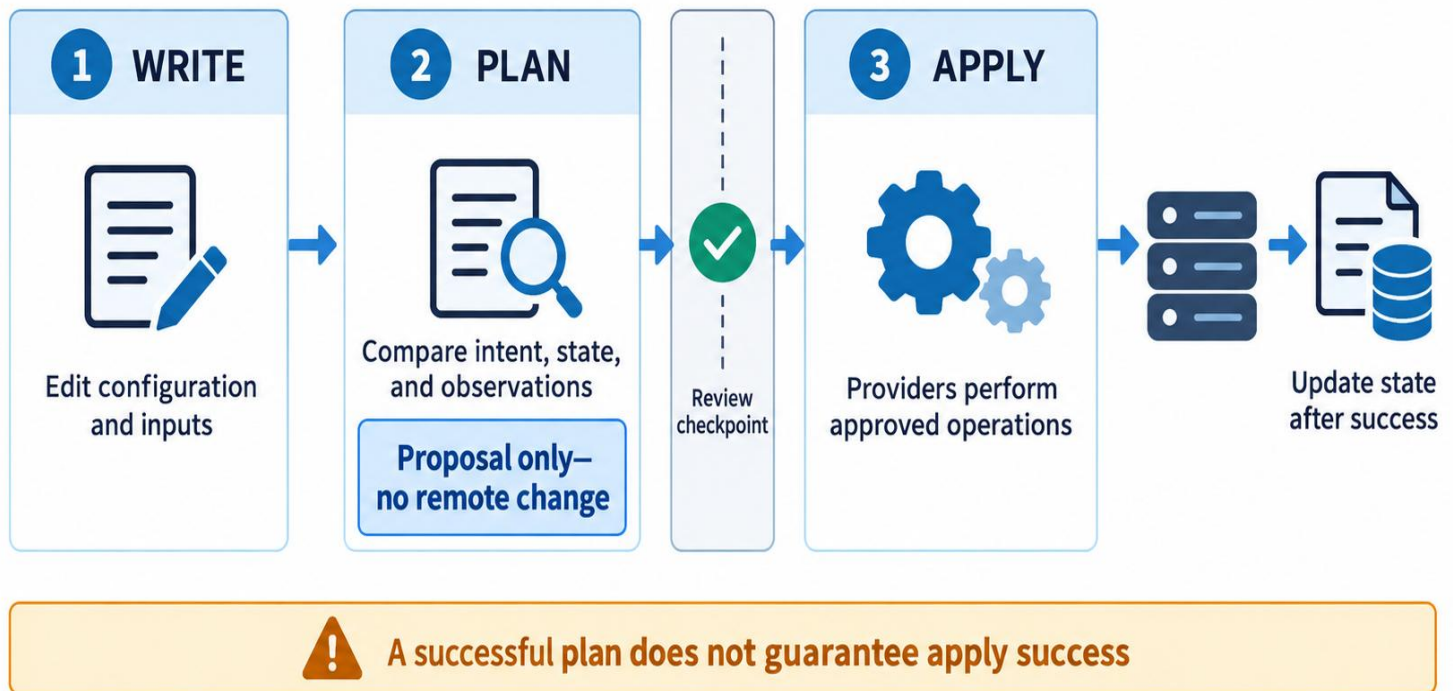
## Apply

An **apply** asks Terraform to perform the planned operations. Providers call their target APIs, Terraform records successful results in state, and the final output reports what completed.

A successful plan does not guarantee a successful apply. Between planning and applying, credentials can expire, quotas can be reached, policies can change, another operator can modify an object, or an API can become unavailable.

# Chapter 1: Terraform in One Mental Model

## 1-4 Write–Plan–Apply Lifecycle



003

Figure 1-3: Write–Plan–Apply Lifecycle

## Terraform follows dependencies, not file order

Terraform builds a **resource graph**, which is a directed graph of objects and the dependencies between them. When one resource argument refers to another resource's attribute, Terraform usually infers the required ordering automatically.

Suppose a subnet contains this reference:

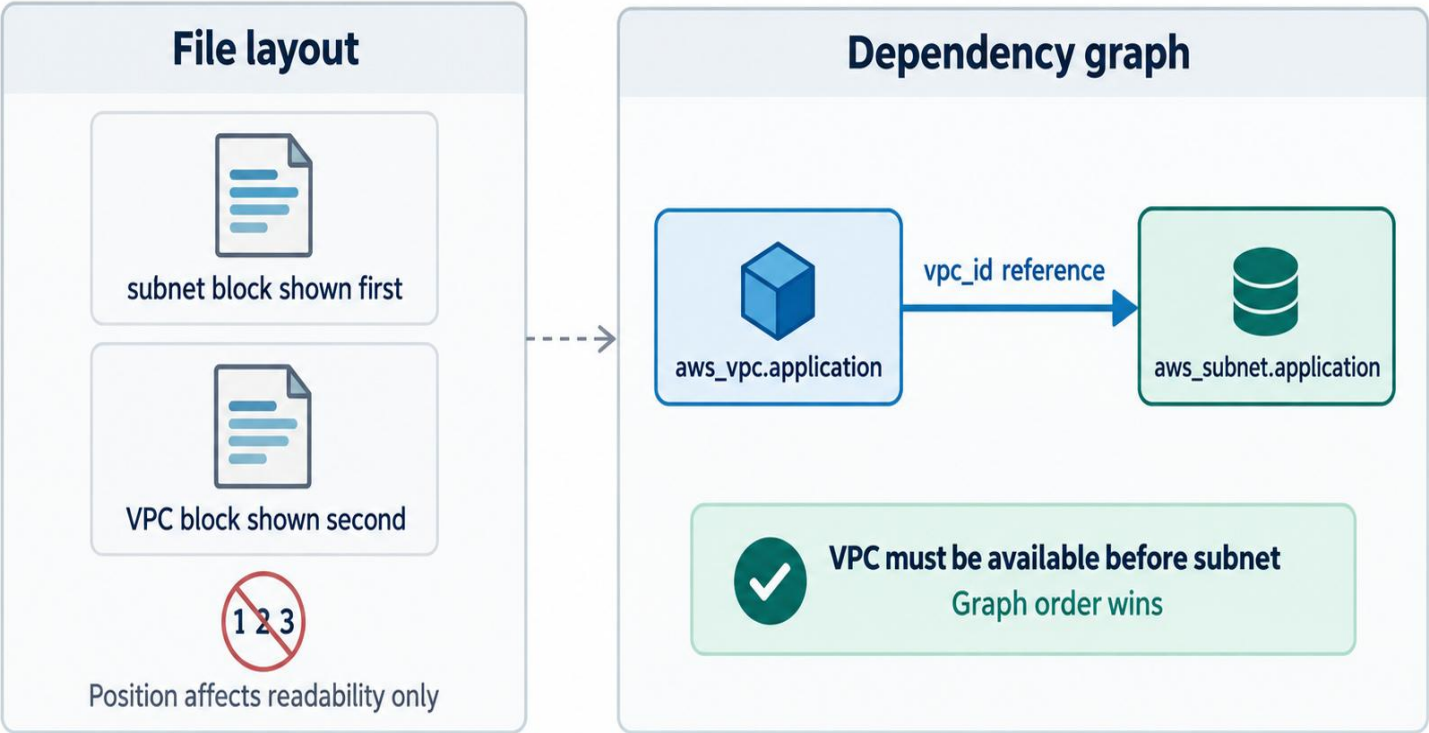
```
vpc_id = aws_vpc.application.id
```

The reference tells Terraform that the VPC must be available before the subnet can be created. Terraform can still create unrelated objects in parallel.

The order of blocks in a file and the alphabetical order of `.tf` files do not control creation order. Reordering files may improve readability, but it does not create a dependency.

# Chapter 1: Terraform in One Mental Model

## 1-5 Dependency Ordering, Not File Ordering



004

Figure 1-4: Dependency Ordering, Not File Ordering

### Desired configuration, recorded state, and actual infrastructure

The three views can agree or disagree:

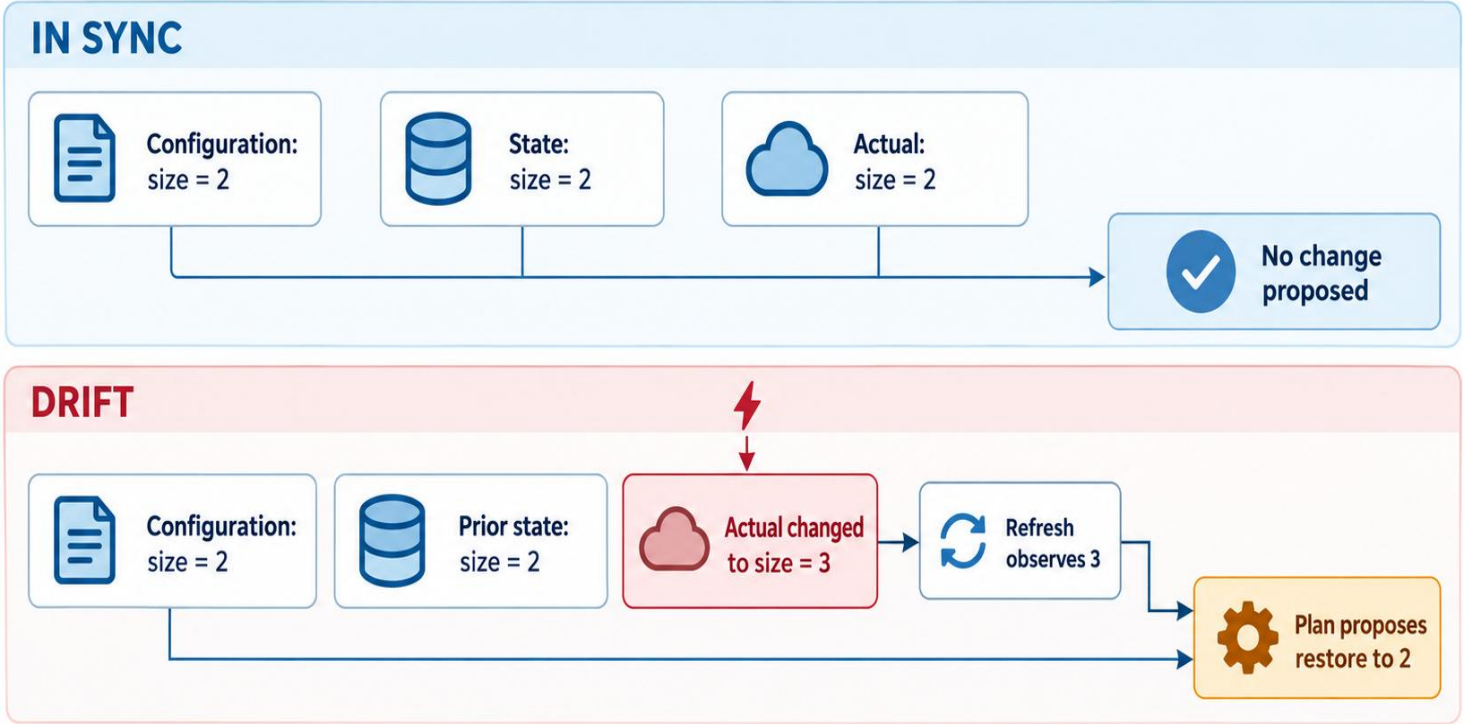
| View                  | What it represents                                                  | Typical source                                    |
|-----------------------|---------------------------------------------------------------------|---------------------------------------------------|
| Desired configuration | What should exist and how it should be configured                   | Versioned <code>.tf</code> files and input values |
| Recorded state        | What Terraform believes it manages and the last recorded attributes | Local or remote state storage                     |
| Actual infrastructure | What currently exists in the target platform                        | Cloud, platform, or SaaS APIs                     |

A difference between actual infrastructure and recorded state is often called **drift**. Drift can result from a manual console change, another automation system, provider defaults, or service-side mutation.

Terraform normally refreshes its understanding of managed objects while planning. It can then propose either changing the remote object to match configuration or updating state to reflect an intentional external change, depending on the chosen workflow.

# Chapter 1: Terraform in One Mental Model

## 1-6 Agreement and Drift Among Three Views



Drift is a difference in remote reality

005

Figure 1-5: Agreement and Drift Among Three Views

### What Terraform manages well

Terraform is well suited to long-lived objects exposed through stable APIs, including:

- Cloud networks, compute instances, databases, and storage services.
- Identity roles, policies, groups, and service accounts.
- Managed platform services and cluster infrastructure.
- DNS records, monitoring objects, and source-control settings.
- SaaS configuration that has a reliable Terraform provider.

Terraform is only one part of an infrastructure toolchain. Other tools may be better suited to different responsibilities:

| Responsibility                                     | Common tool category                            |
|----------------------------------------------------|-------------------------------------------------|
| Building machine or container images               | Image-building tools                            |
| Configuring packages and operating-system settings | Configuration-management tools                  |
| Deploying application releases                     | Deployment systems or continuous-delivery tools |
| Scheduling and healing runtime workloads           | Orchestrators                                   |
| Provisioning durable API-managed infrastructure    | Terraform                                       |

These boundaries are not universal. A team should assign each object to one clear owner and avoid having two systems continually overwrite each other.

## Terraform manages declared ownership, not an entire account

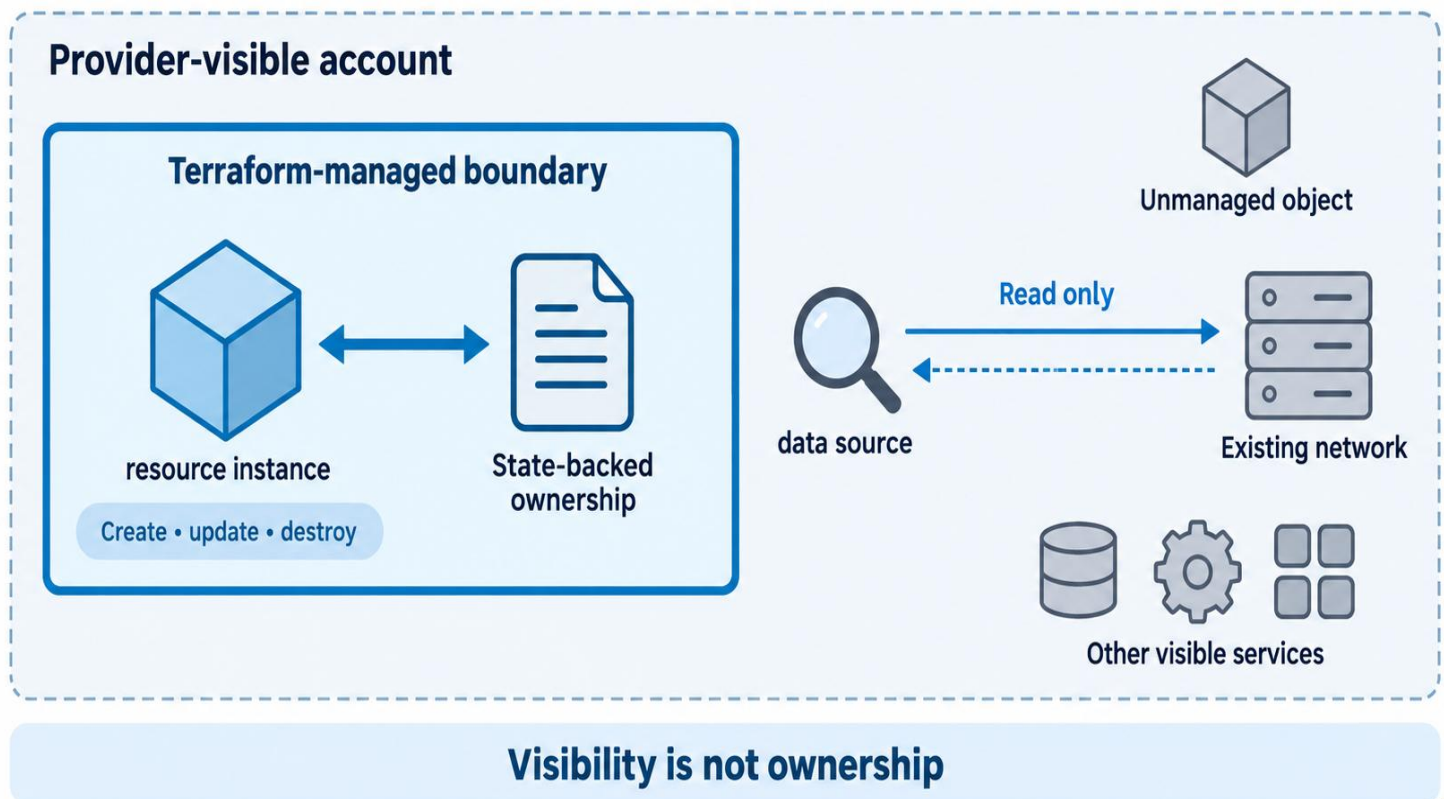
Terraform does not automatically manage every object it can see. A provider may have permission to list thousands of resources, but Terraform manages only objects represented by resource instances in its state.

A **data source** can read an existing object without making Terraform its owner. An unmanaged object can also be adopted later through an import workflow, but simply mentioning its identifier does not transfer ownership.

This distinction matters during deletion. Terraform destroys a managed object when configuration and lifecycle rules call for its removal. It does not destroy an unrelated object merely because the provider can access it.

# Chapter 1: Terraform in One Mental Model

## 1-8 Visibility Versus Terraform Ownership



006

Figure 1-6: Visibility Versus Terraform Ownership

## The practical value

Terraform's main benefits are operational rather than syntactic:

- **Repeatability:** the same configuration can be evaluated consistently.
- **Reviewability:** code and plans expose intended changes before execution.
- **Traceability:** version control records why configuration changed.
- **Dependency handling:** references express ordering without hand-written sequencing.
- **Controlled change:** state and plans let Terraform reason about updates instead of creating everything from scratch.

These benefits depend on disciplined use. Unreviewed plans, exposed state, broad credentials, and competing automation can still cause serious incidents.

## Three misconceptions to remove immediately

**“Terraform runs files from top to bottom.”** It does not. Terraform uses references and explicit dependency declarations to build a graph.

**“Terraform manages everything in the account.”** It does not. It manages resource instances connected to state, plus any new instances declared for creation.

**“A successful plan means apply cannot fail.”** It does not. A plan is based on the information and permissions available at planning time.

The working rule is simple: **treat configuration as intent, state as the ownership map, the plan as a change proposal, and the remote system as the place where operations ultimately succeed or fail.**

## Chapter Summary

**Terraform coordinates declared intent, recorded ownership, and observed infrastructure.** Providers connect Terraform to external APIs, state maps Terraform addresses to remote identities, and plans expose proposed reconciliation actions for review. A correct mental model prevents the common mistakes of confusing visibility with ownership, plan with execution, or file order with dependency order.

## Review Questions

1. What three views does Terraform compare when it plans a change?
2. How do Terraform Core and a provider differ in responsibility?
3. Why does moving a resource block to another `.tf` file normally not change creation order?
4. What must exist before Terraform treats a remote object as managed ownership rather than merely visible data?

## Chapter 2 — A Safe First Terraform Run

### What You'll Be Able to Do

- Run the complete `init` → `fmt` → `validate` → `plan` → `apply` → `destroy` lifecycle in a disposable directory.
- Explain the prerequisite, purpose, and expected result of each core command.
- Recognize configuration, dependency-lock, working-directory, state, and saved-plan artifacts.
- Keep generated files and sensitive operational data out of version control while committing reproducibility inputs.

### Complete the full lifecycle before touching shared infrastructure

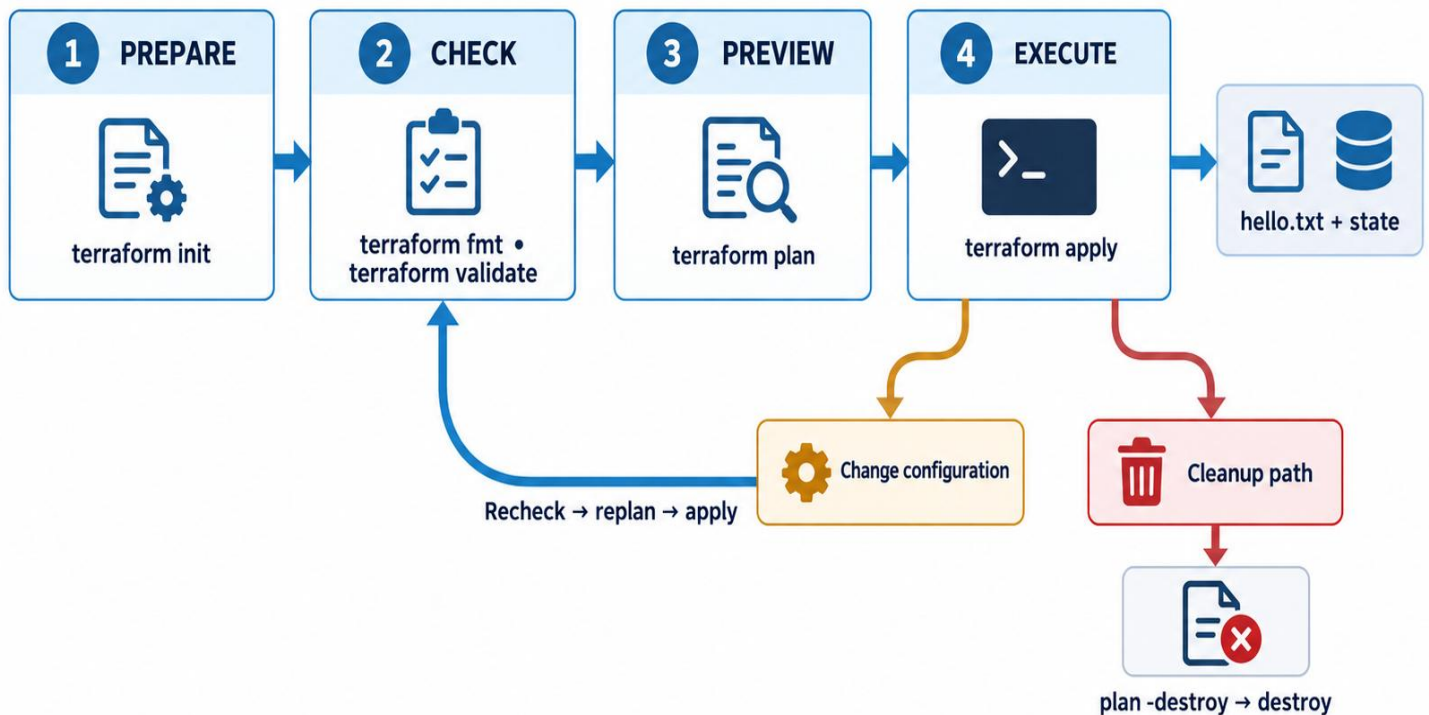
The safest first exercise is a disposable local resource that lets you practice initialization, formatting, validation, planning, applying, changing, and destroying without a cloud account. The goal is not the local file itself. The goal is to see Terraform's operating cycle end to end.

This exercise uses the HashiCorp Local provider to manage one text file. It requires:

- A supported Terraform 1.x executable available on the command path.
- Permission to create files in a new working directory.
- Network access to the Terraform Registry when the provider is downloaded for the first time.

## Chapter 2: A Safe First Terraform Run

### 2-1 Complete Disposable Learning Lifecycle



Practice the full loop on disposable infrastructure

007

Figure 2-1: Complete Disposable Learning Lifecycle

### Verify the executable

**Prerequisite:** Terraform has been installed from an approved source.

Run:

The command prints the Terraform CLI version and platform. A successful result confirms that the shell can find the executable. If a project declares an allowed Terraform version range, the installed version must satisfy that constraint.

## Create an isolated working directory

A **working directory** is the directory Terraform treats as the root module when commands run there. Create an empty directory and change into it. The exact shell commands depend on the operating system.

In that directory, create this configuration:

```
terraform {
  required_version = ">= 1.5, < 2.0"

  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.9"
    }
  }
}

resource "local_file" "welcome" {
  filename = "${path.module}/hello.txt"
  content  = "Terraform created this file.\n"
}

output "welcome_file" {
  value = local_file.welcome.filename
}
```

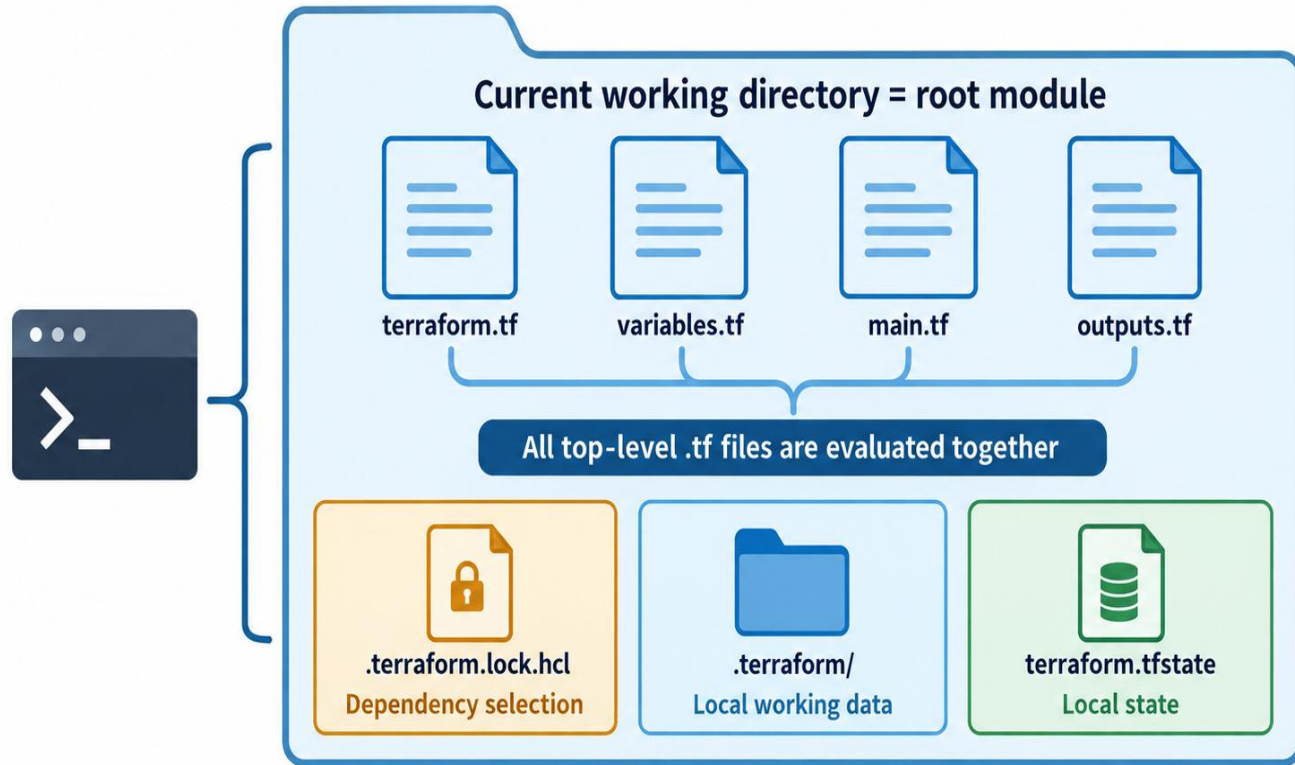
The important lines are:

- `required_version` states which Terraform CLI versions this configuration accepts.
- `required_providers` identifies the Local provider and limits it to compatible 2.x releases beginning with 2.9.
- `resource "local_file" "welcome"` declares one managed local file.
- `path.module` evaluates to the current module's directory.
- The output exposes the resulting file path after apply.

All `.tf` files in this directory form one **root module**, which is the top-level module where Terraform is run. The file name is for human organization; Terraform evaluates the directory as one configuration.

# Chapter 2: A Safe First Terraform Run

## 2-3 Working Directory as the Root-Module Boundary



008

Figure 2-2: Working Directory as the Root-Module Boundary

### Initialize the directory

**Prerequisites:** the configuration exists, the shell is in its directory, and the registry is reachable.

Run:

```
| terraform init
```

`terraform init` prepares the working directory. For this configuration, it downloads a compatible Local provider, creates local initialization data under `.terraform/`, and creates or updates `.terraform.lock.hcl` with the selected provider version and checksums.

The expected result ends with a successful initialization message. Initialization does not create `hello.txt` and does not apply infrastructure changes.

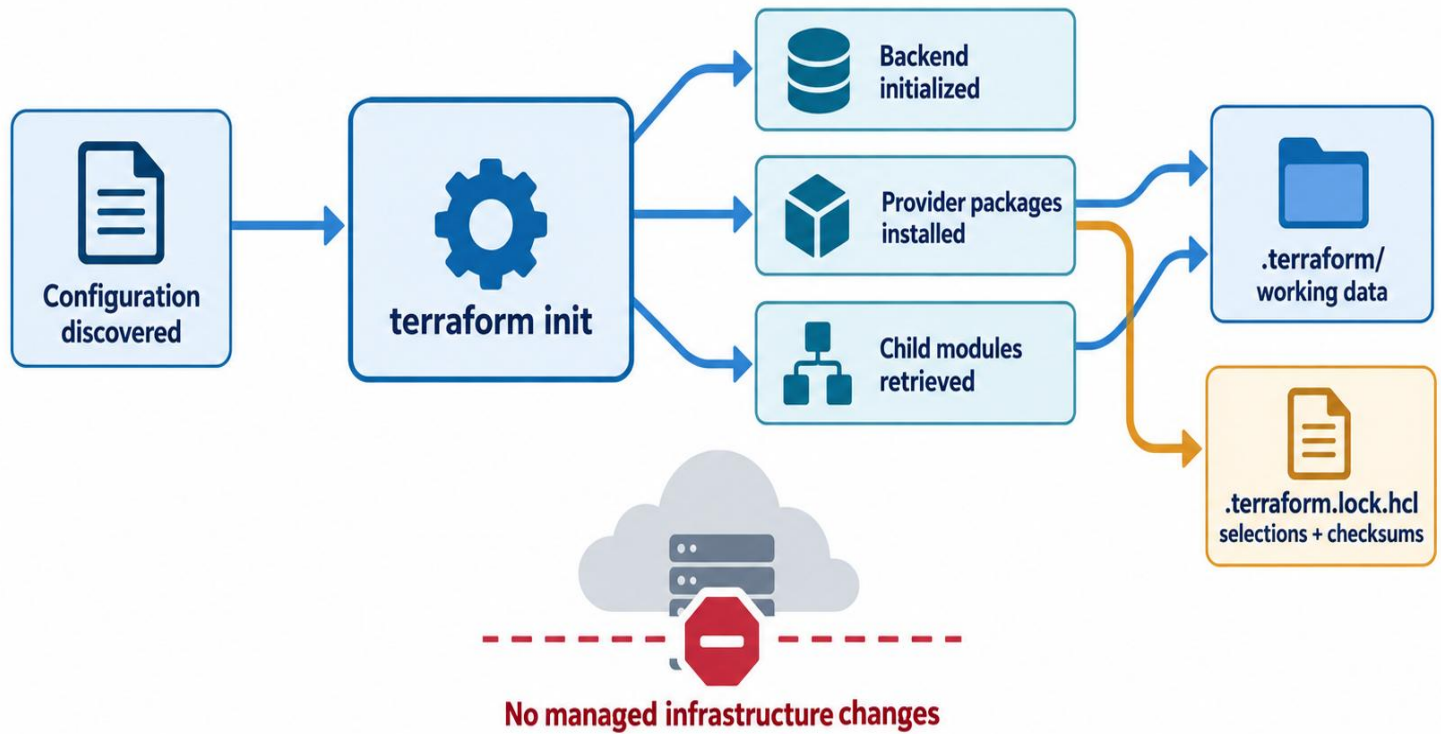
Two generated items now have different purposes:

- `.terraform/` is local working data and downloaded dependencies. It can normally be recreated by running `terraform init` again.
- `.terraform.lock.hcl` is the **dependency lock file**, which records selected provider versions and checksums so repeated runs use consistent provider packages.

Commit the dependency lock file. Do not commit the `.terraform/` directory.

# Chapter 2: A Safe First Terraform Run

## 2-4 What terraform init Prepares



Preparation, not deployment

009

Figure 2-3: What terraform init Prepares

### Format and validate before planning

Run formatting first.

**Prerequisite:** Terraform configuration exists in the current directory.

```
| terraform fmt
```

`terraform fmt` rewrites Terraform files into the standard style. A successful run may print the names of files it changed; no output usually means the files were already formatted.

Then validate the configuration.

**Prerequisite:** the directory has been initialized so provider schemas are available.

```
| terraform validate
```

`terraform validate` checks syntax and internal consistency, including whether references and provider arguments are structurally valid. The expected result states that the configuration is valid.

Validation does not prove that credentials, quotas, policies, or remote API behavior will permit an apply. It is a configuration check, not a deployment guarantee.

### Preview the first change

**Prerequisites:** initialization and validation have succeeded.

```
| terraform plan
```

Terraform reads the configuration and current state. Because no state exists for `local_file.welcome`, the plan should propose creating one resource. Near the end, the summary should be equivalent to:

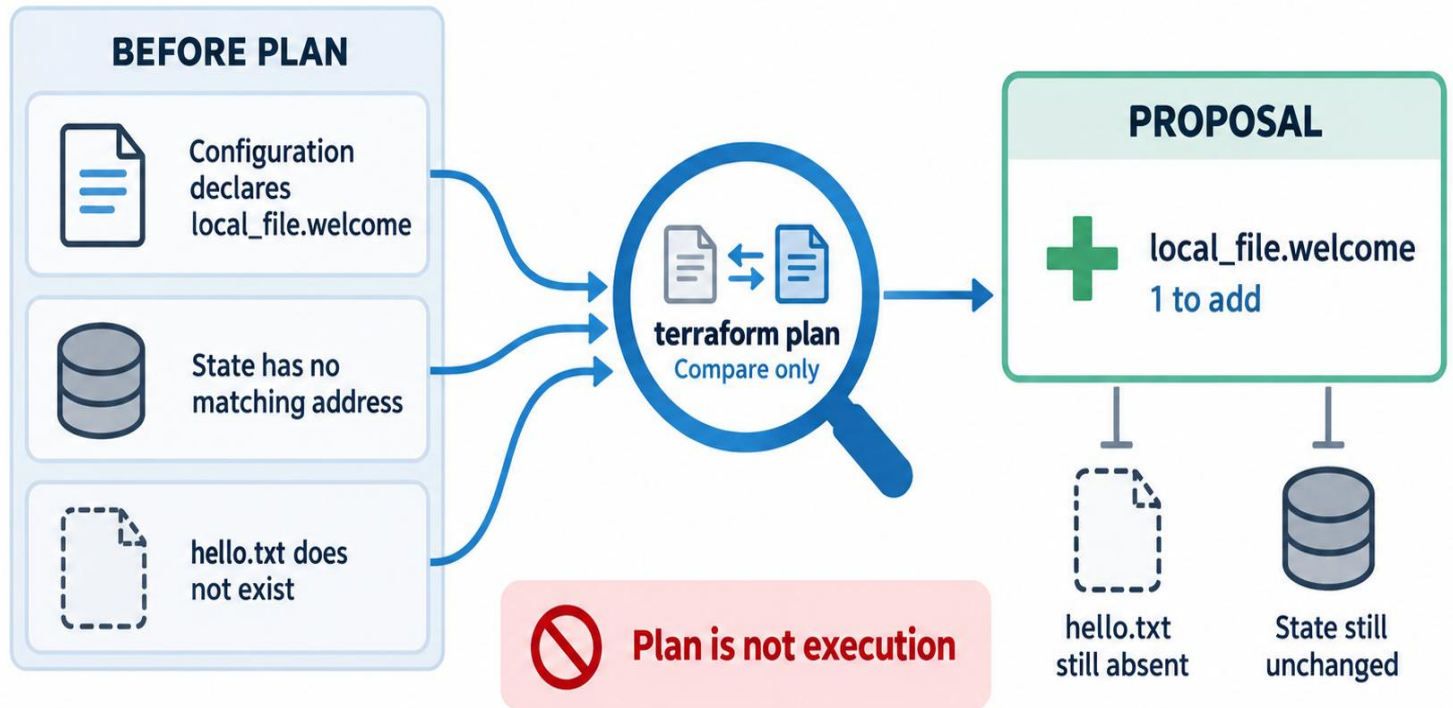
```
| Plan: 1 to add, 0 to change, 0 to destroy.
```

The + symbol marks a planned creation. Values such as a generated identifier may appear as `(known after apply)`, meaning Terraform cannot determine them until the provider completes the operation.

**Planning changes nothing in this exercise.** Read the full resource diff, not only the summary.

## Chapter 2: A Safe First Terraform Run

### 2-6 Why the First Plan Proposes Creation



010

Figure 2-4: Why the First Plan Proposes Creation

### Apply the configuration

**Prerequisite:** the displayed plan contains only the intended local-file creation.

```
| terraform apply
```

Without a saved plan file, `terraform apply` calculates a fresh plan, displays it, and asks for confirmation. Enter `yes` only after checking that the proposed action is still correct.

The expected result reports one resource added and prints the `welcome_file` output. The directory now contains `hello.txt` with this content:

```
| Terraform created this file.
```

Terraform also creates local state, normally named `terraform.tfstate`. The state connects the address `local_file.welcome` to the file managed by the provider.

## Chapter 2: A Safe First Terraform Run

### 2-7 Apply Creates the Object and Records Identity

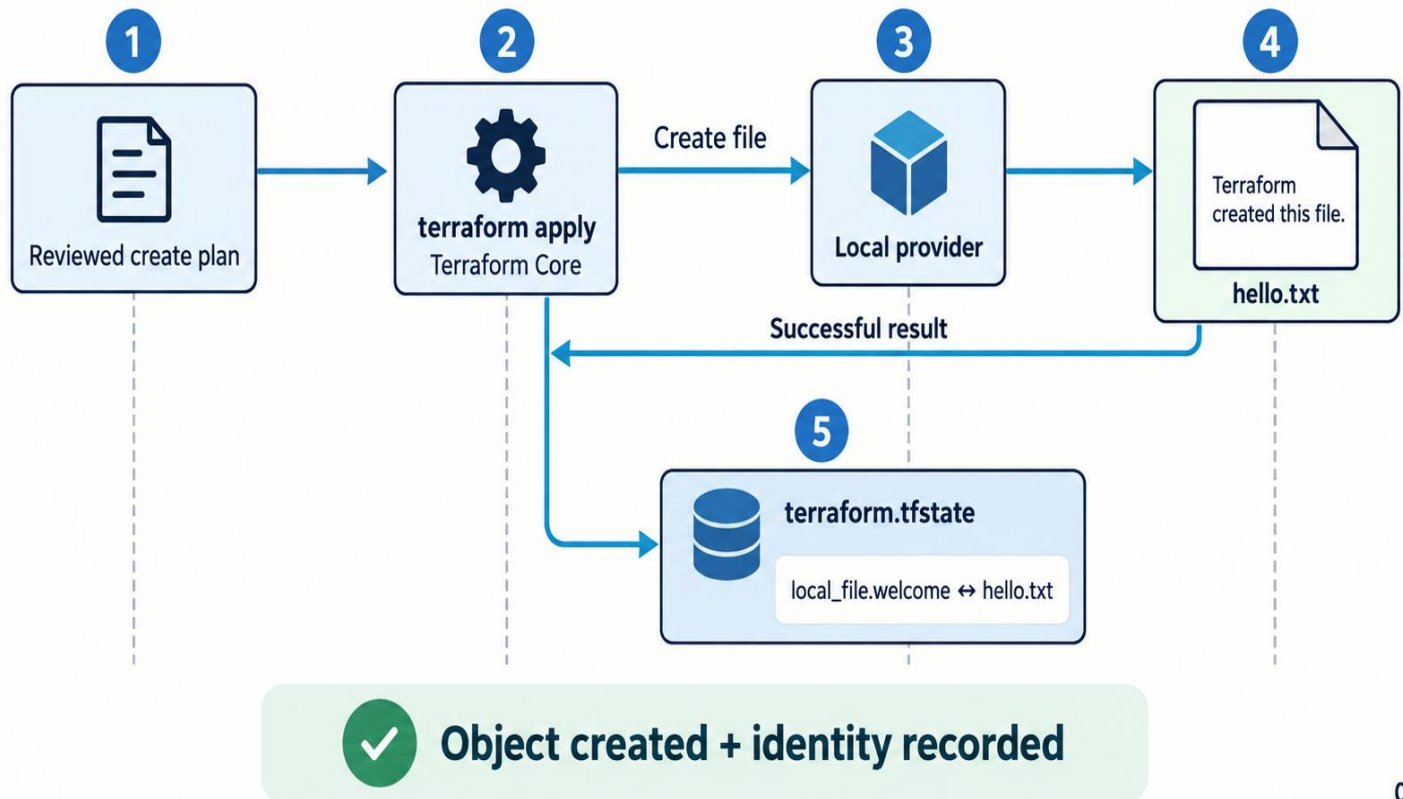


Figure 2-5: Apply Creates the Object and Records Identity

#### Make one controlled change

Change the resource content to:

```
| content = "Terraform updated this file.\n"
```

Format and validate again, then run:

```
| terraform plan
```

Terraform should propose changing the existing managed resource rather than creating an unrelated second resource. The exact representation of the update is provider-specific, but the plan should clearly show the old and new content.

After reviewing the plan, apply it:

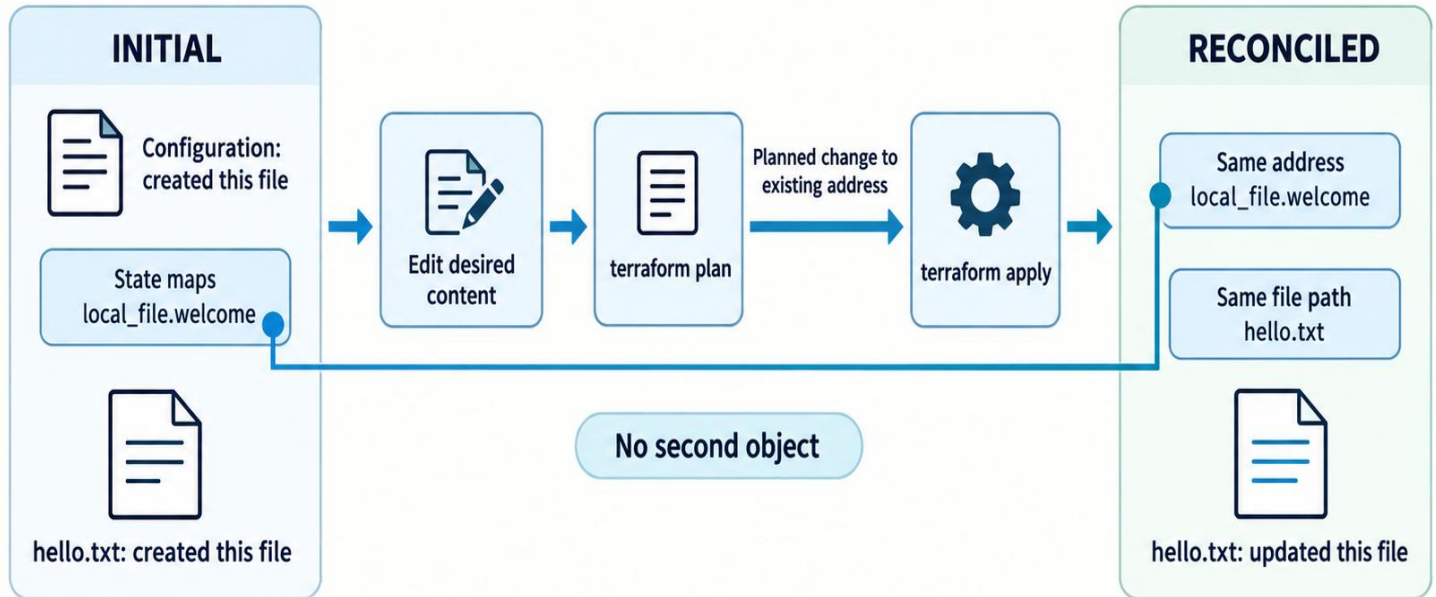
```
| terraform apply
```

The expected result is an updated `hello.txt` and state that records the current resource attributes.

This edit demonstrates reconciliation. You did not write a command that opens a file and replaces text. You changed the desired configuration, and Terraform selected the provider operation needed to make the managed object match it.

## Chapter 2: A Safe First Terraform Run

### 2-8 Controlled Reconciliation After a Configuration Edit



012

Figure 2-6: Controlled Reconciliation After a Configuration Edit

#### Clean up deliberately

**Prerequisite:** the current state still represents the disposable file and the destroy plan affects nothing else.

First preview destruction:

```
| terraform plan -destroy
```

The expected result proposes deleting one managed resource. Then run:

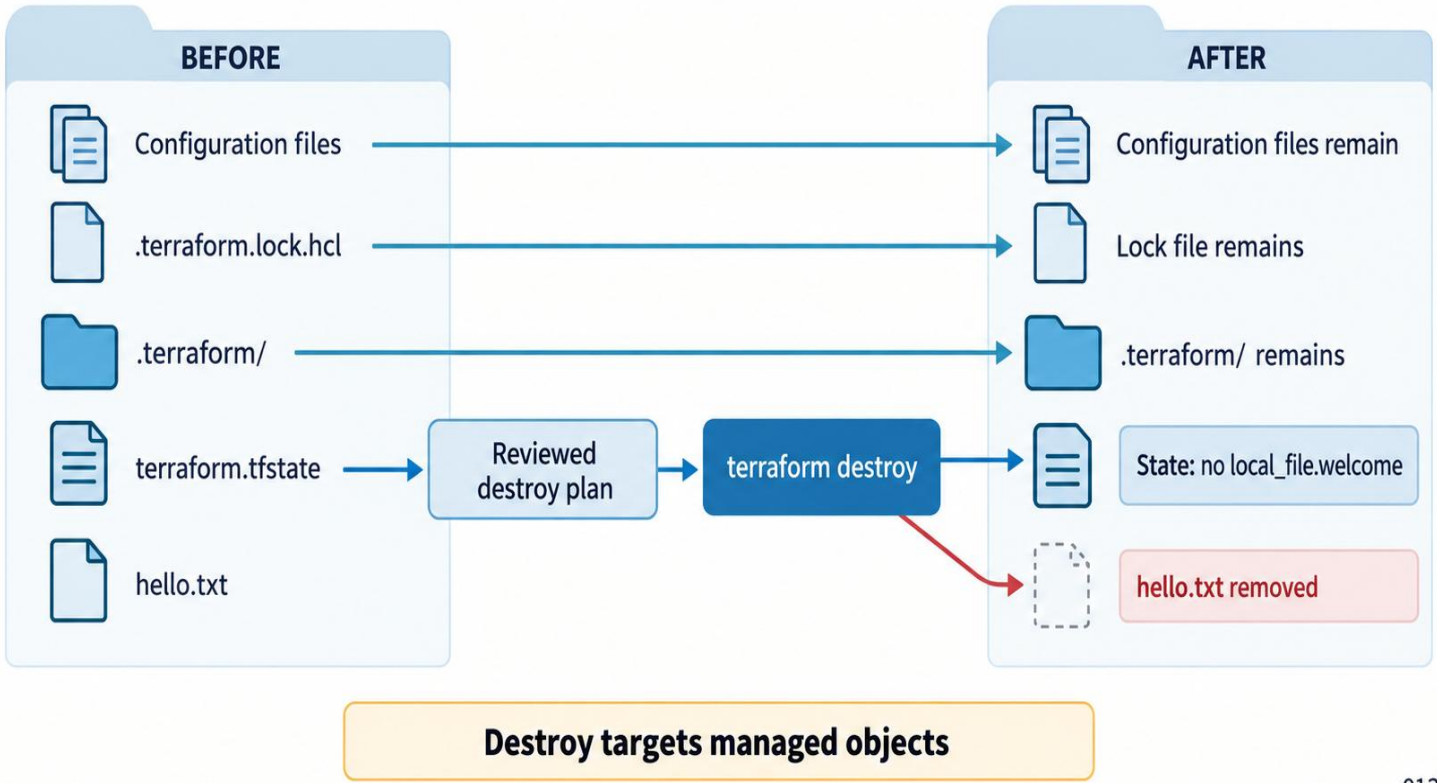
```
| terraform destroy
```

Terraform creates a destroy plan, asks for confirmation, and removes the file after approval. The expected result reports one resource destroyed.

Destroying the resource does not normally delete the configuration, lock file, or initialization directory. It updates state to record that the managed resource no longer exists.

# Chapter 2: A Safe First Terraform Run

## 2-9 Destroy Removes the Managed Object, Not the Project



013

Figure 2-7: Destroy Removes the Managed Object, Not the Project

### Know which generated files matter

| Item                          | Purpose                                                | Version-control treatment |
|-------------------------------|--------------------------------------------------------|---------------------------|
| *.tf                          | Desired configuration                                  | Commit                    |
| .terraform.lock.hcl           | Selected provider versions and checksums               | Commit                    |
| .terraform/                   | Downloaded providers and local initialization data     | Ignore                    |
| terraform.tfstate and backups | Resource mappings and stored attributes                | Do not commit             |
| Saved plan files              | Executable snapshots that may include sensitive values | Do not commit             |
| Secret variable files         | Local secret inputs                                    | Do not commit             |

A small `.gitignore` for this exercise can include:

```
.terraform/  
*.tfstate  
*.tfstate.*  
*.tfplan  
crash.log  
secrets.auto.tfvars
```

A team may intentionally commit non-secret variable definition files. The rule is not “ignore every `.tfvars` file.” The rule is **never place credentials or other secrets in ordinary version control.**

## Generated files are not interchangeable

Deleting `.terraform/` removes recreatable working data. The next command will require initialization again.

Deleting `.terraform.lock.hcl` removes the recorded provider selection and checksums. A later initialization may choose a different allowed provider version, so deleting the lock file casually weakens reproducibility.

Deleting `terraform.tfstate` is fundamentally different. The local file may still exist, but Terraform has lost the mapping that says it manages that object. A future plan can attempt to create another object, encounter a name collision, or leave the existing object unmanaged.

**Never delete state merely to make an error disappear.** Understand the state and the remote object first.

## What this first run proves

This exercise establishes the basic operational loop:

1. Configuration states the desired object.
2. Initialization prepares providers and dependency selections.
3. Formatting and validation catch basic configuration problems.
4. Planning previews the proposed reconciliation.
5. Applying performs approved operations and updates state.
6. Destroying is also a planned, stateful operation.

It also establishes the first repository rule: **commit configuration and the dependency lock file; protect state, plans, provider downloads, and secret inputs.**

## Chapter Summary

**A safe first run teaches the complete operating cycle before shared infrastructure is involved.** Initialization installs dependencies, formatting and validation catch local problems, planning previews actions, apply performs approved work, and destroy removes the disposable result. Repository hygiene matters from the first exercise because state, downloaded plugins, saved plans, and secret values have different security and reproducibility roles.

## Review Questions

1. What does `terraform init` prepare, and when must it be run again?
2. Why should `terraform plan` be reviewed even for a small configuration?
3. Which generated artifact should normally be committed: `.terraform/`, `terraform.tfstate`, or `.terraform.lock.hcl`?
4. What should you verify before running `terraform destroy`?