THE DEFINITIVE REFERENCE

OPEN TOFU

INFRASTRUCTURE AS CODE

FOR MODERN CLOUD AND ON-PREMISES ENVIRONMENTS

DEFINE
YOUR INFRASTRUCTURE

PROVISION
ANYWHERE

MANAGE
WITH CONFIDENCE

SECURE
AT SCALE

FROM FIRST PRINCIPLES TO ENTERPRISE-SCALE PRODUCTION DEPLOYMENTS

MICHAEL BROOKS

OpenTofu: The Definitive Reference

Infrastructure as Code for Modern Cloud and On-Premises Environments

Steve Publications

This book is available at
https://leanpub.com/opentofuthedefinitivereference

This version was published on 2026-08-03

Contents

Infrastructure as Code for Modern Cloud and On-Premises Environments

OpenTofu is the open-source, community-driven infrastructure-as-code tool that lets you define, provision, and manage cloud and on-premises resources using declarative configuration. This book takes you from first principles through enterprise-scale production deployments, covering every aspect of OpenTofu with working examples, architectural guidance, and real-world patterns used by leading DevOps teams. Whether you are migrating from Terraform or learning infrastructure-as-code for the first time, this reference will become your go-to resource for building reliable, maintainable, and secure infrastructure at scale.

Chapter 1: The OpenTofu Story: From Terraform to Open Source

What Is OpenTofu?

OpenTofu is an open-source infrastructure-as-code tool that lets you define, provision, and manage cloud and on-premises resources using declarative configuration files written in HCL (HashiCorp Configuration Language). It operates as a drop-in replacement for Terraform, preserving the same workflow, syntax, provider ecosystem, and state file format while returning to genuinely open-source licensing under the Mozilla Public License 2.0.

At its core, OpenTofu solves a fundamental problem: how do you manage complex, distributed infrastructure in a consistent, repeatable, and auditable way? The answer is to describe your desired infrastructure state in code, then let OpenTofu figure out what changes are needed to make reality match that description. You write configuration files that declare what resources you want: virtual machines, load balancers, databases, IAM policies, Kubernetes clusters. OpenTofu communicates with the underlying cloud providers or on-premises platforms through plugins called providers to create, update, or destroy those resources.

Key characteristics of OpenTofu include:

- **Declarative syntax**: You describe what infrastructure you want, not how to build it step by step.
- **Provider-based architecture**: Thousands of community and vendor-maintained providers let OpenTofu manage resources across AWS, Azure, GCP, Kubernetes, GitHub, Datadog, and hundreds of other platforms.
- **State management**: OpenTofu tracks the real-world state of your infrastructure in a state file, enabling it to compute accurate plans for changes and detect drift.
- **Modular design**: Reusable modules let you package common infrastructure patterns and share them across teams and projects.

- **Plan-before-apply workflow**: Every change is previewed in a plan that shows exactly what will be created, modified, or destroyed before any action is taken.

As of mid-2026, OpenTofu has reached version 1.12.x with active development, over 29,000 GitHub stars, and more than 70 active contributors. It is governed by the Linux Foundation through a Technical Steering Committee drawn from multiple organizations, ensuring it cannot be closed off or controlled by a single vendor.

The Terraform License Change: A Turning Point

To understand why OpenTofu exists, you need to understand what happened with Terraform.

For over a decade, Terraform was the dominant infrastructure-as-code tool in the industry. Released by HashiCorp in 2014 and open-sourced under the Mozilla Public License 2.0 (MPL 2.0), it became the default choice for DevOps teams worldwide. Its provider ecosystem grew to thousands of plugins, its module registry hosted tens of thousands of reusable configurations, and entire organizations built their platform engineering practices around Terraform workflows.

Then on August 10, 2023, HashiCorp announced a sweeping change: all future releases of Terraform and its other products would switch from the MPL 2.0 open-source license to the Business Source License (BSL) 1.1, also known as BUSL. This was not a minor adjustment. The BSL is a source-available but non-open-source license that restricts how the software can be used commercially: specifically, it prohibits any entity from offering a competing hosted service based on the software without HashiCorp's permission.

The motivation behind the change was clear from HashiCorp's announcement: cloud providers like AWS and Google were offering managed Terraform services (AWS CloudFormation with Terraform integration, Google's Terraform workspaces) that competed directly with HashiCorp's own commercial offerings. Under MPL 2.0, those competitors could legally fork Terraform and build their own commercial products on top of it. The BSL was designed to prevent exactly that.

The community reaction was swift and overwhelmingly negative. Within days:

- A manifesto calling for a return to open-source licensing gathered over 33,000 GitHub stars and the support of nearly 140 companies.
- Major platform engineering tools like Spacelift, Env0, Scalr, and many others publicly announced they would not comply with BSL restrictions and would fork Terraform instead.
- Open-source advocates raised concerns that HashiCorp had broken an implicit trust: teams had built their infrastructure automation around a tool they believed would remain open-source in perpetuity.

The core issue was philosophical as much as commercial. Infrastructure-as-code tools sit at the foundation of how organizations build and operate software systems. When that foundation is controlled by a single vendor under a license that can change direction without community input, it creates strategic risk for every team depending on it.

HashiCorp vs. Linux Foundation: The Fork That Changed Everything

The response to the license change was not just criticism; it was action. Within weeks of the announcement, work began on OpenTF (later renamed OpenTofu), a fork of Terraform based on the last MPL-licensed version: 1.5.x.

The fork had clear goals from the start:

1. **Maintain compatibility**: Existing Terraform configurations should work without modification.
2. **Preserve open-source licensing**: The project would remain under MPL 2.0 forever.
3. **Ensure vendor neutrality**: Governance would be multi-stakeholder, not controlled by any single company.

On September 20, 2023, at the Open Source Summit in Bilbao, the Linux Foundation announced that OpenTF had been accepted as a project under its stewardship – with a name change to OpenTofu due to trademark considerations (HashiCorp retained ownership of the Terraform trademark). The rename was handled pragmatically: just as PostgreSQL forked from Postgres while keeping similar naming conventions, OpenTofu would be recognizable as related while establishing its own identity.

The technical steering committee for OpenTofu included representatives from multiple organizations including Spacelift, Env0, and others that had been early supporters of the fork. This multi-vendor governance model was deliberately chosen to mirror how projects like Kubernetes are managed under the Cloud Native Computing Foundation: no single entity can unilaterally change direction or close the project.

Development moved quickly. Within months:

- **December 2023**: Release candidates were published, with stability improvements and a new public registry for providers and modules at search.opentofu.org.
- **January 10, 2024**: OpenTofu 1.6 was released as the first general availability version, maintaining full backward compatibility with Terraform 1.5.x.
- **April 2024**: OpenTofu 1.7 introduced the first major feature divergence: native state encryption at rest, protecting sensitive infrastructure data without relying on backend-specific encryption features.
- **Mid-2024 onward**: Features like early variable evaluation (v1.8), provider iteration with for_each (v1.9), OCI registry support (v1.10), and dynamic prevent_destroy (v1.12) marked OpenTofu's evolution beyond mere compatibility into independent innovation.

By mid-2026, adoption metrics from platforms like Scalr showed OpenTofu accounting for approximately 63% of runs and 72% of new workspaces on their platform, indicating a significant shift in the industry toward the open-source alternative.

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

As of mid-2026, OpenTofu and Terraform remain remarkably similar at the day-to-day usage level. Both use HCL configuration syntax, both manage state files in the same format, and both support the same provider ecosystem through the plugin protocol. For most teams, switching from `terraform` to `tofu` in their command line is literally the only change needed.

However, meaningful differences exist across several dimensions:

Licensing and governance

This is the fundamental distinction. OpenTofu is licensed under MPL 2.0, an OSI-approved open-source license that guarantees anyone can use, modify, and distribute the software for any purpose without restrictions. Terraform (since v1.6) uses BSL 1.1, which is source-available but not open-source: you can view and run the code, but commercial entities cannot offer competing hosted services based on it.

Governance differs equally dramatically. OpenTofu is stewarded by the Linux Foundation through a Technical Steering Committee with representatives from multiple organizations, ensuring balanced decision-making. Terraform is owned and controlled by HashiCorp (which became an IBM subsidiary in February 2025), with all strategic decisions made internally.

Feature divergence

While both tools share the same core capabilities, OpenTofu has introduced several features that Terraform does not have:

- **Native state encryption (v1.7)**: Encrypt state files at rest using PBKDF2, AWS KMS, or GCP KMS keys, configurable via environment variables or CLI flags.
- **Early variable evaluation (v1.8)**: Variables are evaluated earlier in the planning process, enabling more flexible module interfaces and reducing certain class of "depends on apply" errors.
- **Provider for_each (v1.9)**: Dynamically create provider configurations from collections, greatly simplifying multi-region and multi-account patterns.
- **Resource exclusion flag (v1.9)**: The `-exclude` flag lets you skip specific resources during plan and apply operations without modifying configuration.
- **OCI registry support (v1.10)**: Publish and consume providers and modules from OCI-compatible registries, not just the OpenTofu Registry.
- **Dynamic prevent_destroy (v1.12)**: The `prevent_destroy` lifecycle argument now accepts variable expressions, enabling environment-specific protection without maintaining separate module copies.

Conversely, Terraform has features that OpenTofu does not replicate:

- **Terraform Stacks**: A HashiCorp Cloud Platform feature for orchestrating cross-workspace dependencies using a higher-level abstraction.

- **Certain HCP Terraform integrations**: Tight coupling with HashiCorp's commercial cloud platform.

Compatibility details

OpenTofu 1.6 was designed to be fully compatible with Terraform configurations up through version 1.5.x. Most configurations written for later Terraform versions also work, though features introduced after the fork point may not be supported. Provider compatibility is excellent: the vast majority of providers published to the HashiCorp Registry work identically with OpenTofu because they communicate via a well-defined plugin protocol that both tools implement.

State file format is compatible in most cases. When you run `tofu init` on an existing Terraform project, OpenTofu reads the existing state and continues managing your infrastructure without disruption. The only change is that the state file's version metadata will be updated to indicate it was last managed by OpenTofu.

When each makes sense

Choose OpenTofu when:

- You want open-source guarantees that cannot be changed unilaterally.
- You operate a commercial platform or service built on top of infrastructure-as-code tooling.
- You prefer multi-vendor governance over single-company control.
- You value the specific features OpenTofu has added (state encryption, provider for_each, etc.).
- Your organization's procurement or legal team requires OSI-approved open-source licensing.

Choose Terraform when:

- You are heavily invested in HashiCorp Cloud Platform and use Stacks or other HCP-specific features.
- You require IBM Cloud Pak integration or similar enterprise partnerships tied to Terraform specifically.
- Your team has standardized on the HashiCorp ecosystem (Vault, Consul, Nomad) and wants unified vendor support.

For most teams in 2026, OpenTofu represents the lower-risk default choice due to its open-source licensing and community governance, while remaining functionally equivalent for daily operations.

When to Choose OpenTofu (and When Not To)

The decision between OpenTofu and Terraform should be driven by your organization's specific needs, risk tolerance, and existing investments. Here is a practical framework:

Strong reasons to choose OpenTofu:

1. **You build commercial products or services**: If you are offering infrastructure-as-code tooling as part of a SaaS product, the BSL restrictions on Terraform could expose you to legal risk. OpenTofu's MPL 2.0 license eliminates this concern entirely.
2. **Open-source policy requirements**: Many organizations have internal policies requiring all tools to be OSI-approved open-source software. Under these policies, Terraform (BSL) does not qualify, while OpenTofu (MPL 2.0) does.
3. **Long-term vendor independence**: If your concern is that a single company controlling your infrastructure tooling creates strategic risk — whether through license changes, pricing increases, or strategic pivots — OpenTofu's multi-vendor governance model directly addresses this.
4. **Feature preferences**: The features OpenTofu has added (native state encryption, provider for_each, dynamic prevent_destroy) are genuinely useful improvements that many teams want regardless of licensing concerns.

Reasons to stick with Terraform:

1. **Deep HCP Terraform investment**: If your organization has invested significantly in HashiCorp Cloud Platform — using Stacks, Sentinel policies, team collaboration features, audit logging, and cost management — migrating those capabilities to OpenTofu would require finding alternative solutions or building them yourself.

2. **Enterprise support contracts**: Some organizations have enterprise support agreements with HashiCorp that cover Terraform specifically. If you rely on this support model, switching to OpenTofu means either ending that contract or finding a different support provider for OpenTofu.
3. **Minimal friction preference**: If your current Terraform setup works perfectly, your team is trained on it, your CI/CD pipelines are built around it, and you have no licensing concerns, there may be no compelling reason to switch. The migration is straightforward but still requires effort.

Migration considerations:

If you decide to migrate from Terraform to OpenTofu, the process is generally smooth:

1. Back up your state files and configuration.
2. Install OpenTofu alongside or instead of Terraform.
3. Run `tofu init` in each project directory – this downloads providers and initializes backends.
4. Run `tofu plan` to verify that the existing infrastructure is recognized correctly with no unexpected changes.
5. Run `tofu apply` to update state file metadata (this typically shows "No changes" for actual infrastructure).
6. Update CI/CD pipelines to use `tofu` instead of `terraform`.
7. Test with a small, non-critical change to validate the full workflow.

The rollback path is equally straightforward: restore your backed-up state files and run `terraform init`. This reversibility makes migration low-risk for most teams.

In the chapters that follow, you will learn OpenTofu from the ground up – installation, configuration syntax, providers, resources, state management, modules, testing, security, CI/CD integration, and enterprise-scale patterns. Whether this is your first infrastructure-as-code tool or you are transitioning from Terraform, the knowledge here applies directly to building production-grade infrastructure that is reliable, maintainable, and secure.

Chapter 2: Architecture and Core Concepts

The Declarative Paradigm: What You Define vs. What Happens

Infrastructure-as-code tools like OpenTofu operate on a fundamentally different model than traditional configuration management or scripting approaches. Understanding this distinction is essential to using OpenTofu effectively.

Traditional scripting is imperative: you write step-by-step instructions telling the system exactly what to do in what order. A bash script that provisions a server might look like this conceptually:

- Connect to the cloud API
- Check if the instance already exists
- If not, create it with these parameters
- Wait for it to be running
- Install software using SSH
- Configure services

Each step must be explicitly handled. You manage errors, retries, and edge cases yourself. The script is a recipe that describes a process.

OpenTofu uses declarative configuration instead. You describe the desired end state of your infrastructure, and OpenTofu figures out what steps are needed to reach that state from wherever you currently are. Your configuration says: "I want an EC2 instance with this AMI, this instance type, in this subnet, with these tags." It does not say how to create it, wait for it, or configure it.

This declarative approach has several important consequences:

Idempotency: Running the same OpenTofu configuration multiple times produces the same result. If the infrastructure already matches your desired state, nothing changes. If something drifted (someone manually modified

a resource in the console), OpenTofu detects the difference and proposes corrections. This is dramatically different from imperative scripts that might create duplicate resources or apply configurations repeatedly with unintended side effects.

State awareness: Because OpenTofu tracks what actually exists in your infrastructure, it can compute accurate plans for changes. When you modify a configuration to add a load balancer or change an instance type, OpenTofu compares the new desired state against the current real state and tells you exactly what will happen before you apply any changes.

Drift detection: If someone manually changes a resource outside of OpenTofu – adding a security group rule in the AWS console, for example – the next time you run `tofu plan`, OpenTofu detects that reality no longer matches your configuration and proposes reverting the change (or you can update your configuration to accept it). This prevents "configuration drift" where infrastructure gradually diverges from documented intentions.

The trade-off is that declarative tools require a different mental model. You cannot reason about OpenTofu configurations by reading them top-to-bottom like a script. Instead, you must understand the graph of resources and their dependencies, how state is tracked, and how providers interact with APIs. The rest of this chapter establishes that foundation.

OpenTofu's Execution Model: Plan, Apply, Destroy

Every interaction with OpenTofu follows a core workflow cycle: write configuration, initialize, plan changes, apply changes, and eventually destroy resources when they are no longer needed. This workflow is not just a sequence of commands; it is the fundamental design philosophy that makes infrastructure-as-code safe and auditable.

Step 1: Write configuration

You write OpenTofu configuration files with `.tf` extensions in your working directory. These files describe the infrastructure you want using HCL (HashiCorp Configuration Language). A minimal example creating an S3 bucket looks like this:

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

provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "example" {
  bucket = "my-example-bucket-unique-name"
}
```

This configuration declares three things: it requires the AWS provider, it configures that provider to use the us-east-1 region, and it wants an S3 bucket with a specific name. Notice that you do not specify how to create the bucket, authenticate with AWS, or handle errors – those details are abstracted away.

Step 2: Initialize (`tofu init`)

Before OpenTofu can work with your configuration, it must initialize the working directory:

```
$ tofu init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/aws versions matching "~> 5.0"...
- Installing hashicorp/aws v5.89.0...
- Installed hashicorp/aws v5.89.0 (signed by HashiCorp)

Terraform has been successfully initialized!
```

This command performs several critical tasks:

- Downloads and installs provider plugins specified in your configuration.
- Initializes the backend (where state is stored – local file by default, or remote storage like S3 for team collaboration).
- Downloads any remote modules referenced in your configuration.
- Creates a `.terraform` directory containing providers and module metadata.

- Optionally upgrades providers to newer versions within their version constraints if you use `-upgrade`.

You must run `tofu init` every time you start working with a new configuration or add new providers/modules. It is the foundation that everything else builds on.

Step 3: Plan (`tofu plan`)

The plan command is OpenTofu's safety mechanism. Before making any changes to real infrastructure, it computes what those changes would be and shows you in detail:

```
$ tofu plan
Terraform used the selected providers to generate the following execution plan.
↪  Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_s3_bucket.example will be created
  + resource "aws_s3_bucket" "example" {
      + bucket = "my-example-bucket-unique-name"
      + id     = (known after apply)
    }

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

The plan output shows:

- `+ create`: Resources that will be created.
- `~ update`: Resources that will be modified in-place.
- `-/+ replace`: Resources that will be destroyed and recreated (for changes that cannot be applied in-place).
- `- destroy`: Resources that will be deleted.

Critical aspects of the plan phase:

- **It is read-only**: Running `tofu plan` never modifies your infrastructure. It only reads the current state, reads your configuration, computes differences, and displays them.

- **It refreshes state first**: Before computing the plan, OpenTofu queries providers to get the current real-world state of each resource. This ensures the plan is based on accurate information, not stale cached data.
- **You can save plans**: Using `tofu plan -out=tfplan`, you can save a plan as a binary file and later apply that exact plan with `tofu apply tfplan`. This is important for CI/CD workflows where the person reviewing the plan may not be the same person or system applying it.

Step 4: Apply (`tofu apply`)

Once you review and approve a plan, the apply command executes it:

```
$ tofu apply
# ... shows the same plan as above ...
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

aws_s3_bucket.example: Creating...
aws_s3_bucket.example: Creation complete after 2s
↪  [id=my-example-bucket-unique-name]

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

During apply, OpenTofu:

- Acquires a state lock to prevent concurrent modifications (when using remote backends with locking support).
- Creates, updates, or destroys resources in dependency order – resources that other resources depend on are created first.
- Updates the state file to reflect the new infrastructure reality.
- Releases the state lock when complete.

You can apply a saved plan file (`tofu apply tfplan`) which will execute exactly what was planned without re-computing, or run `tofu apply` directly which computes a fresh plan and prompts for approval.

Step 5: Destroy (`tofu destroy`)

When infrastructure is no longer needed, the destroy command removes all resources managed by your configuration:

```
$ tofu destroy
# ... shows plan to destroy all resources ...
aws_s3_bucket.example: Destroying... [id=my-example-bucket-unique-name]
aws_s3_bucket.example: Destruction complete after 1s

Destroy complete! Resources: 1 destroyed.
```

Destroy respects dependency order in reverse — it destroys dependent resources first, then the resources they depend on. This is critical for avoiding errors like trying to delete a VPC while subnets still exist inside it.

This plan-apply-destroy workflow, combined with state management and provider plugins, forms the complete execution model of OpenTofu. Every operation you perform fits into this cycle, making infrastructure changes predictable, reviewable, and reversible.

Providers: The Bridge Between Configuration and Reality

Providers are the mechanism by which OpenTofu communicates with real-world infrastructure. They are plugins that implement APIs for specific platforms or services, translating OpenTofu's abstract resource definitions into concrete API calls.

How providers work

When you declare a resource like `aws_s3_bucket`, OpenTofu does not know anything about Amazon S3 natively. Instead, it delegates to the AWS provider plugin, which knows how to call the AWS API to create an S3 bucket. The provider plugin:

1. Defines what resource types it supports (S3 buckets, EC2 instances, IAM roles, etc.).
2. Specifies what arguments each resource type accepts (bucket name, instance type, policy document).
3. Implements CRUD operations (Create, Read, Update, Delete) for each resource type by calling the appropriate APIs.
4. Reports back to OpenTofu about the current state of resources so that plans can be computed accurately.

This plugin architecture is what makes OpenTofu work across dozens of platforms with a single consistent interface. You write HCL configuration and run `tofu plan` and `tofu apply` regardless of whether you are managing AWS, Azure, GCP, Kubernetes, or any other platform that has a provider.

Provider ecosystem

As of mid-2026, there are over 3,900 providers available in the OpenTofu Registry at search.opentofu.org. These include:

- **Major cloud providers**: AWS (hashicorp/aws), Azure (hashicorp/azurerm), GCP (hashicorp/google), Oracle Cloud, IBM Cloud, DigitalOcean, Linode, Vultr.
- **Container and orchestration platforms**: Kubernetes (kubernetes/kubernetes), Docker (kreuzwerker/docker), Nomad (hashicorp/nomad).
- **Networking and DNS**: Cloudflare (cloudflare/cloudflare), Route53 via AWS provider, NS1, PowerDNS.
- **Databases and data services**: MongoDB Atlas, Redis Enterprise, Snowflake, Databricks.
- **CI/CD and collaboration tools**: GitHub (integrations/github), GitLab, Jira, Slack.
- **Monitoring and observability**: Datadog, Prometheus, Grafana, New Relic.
- **Security and identity**: Okta, Auth0, OneLogin, Ping Identity.
- **Specialized providers**: Null provider for local-only resources, Random provider for generating random values, HTTP provider for calling arbitrary REST APIs, Template provider for string manipulation.

Providers are published to the OpenTofu Registry with versioned releases. Your configuration specifies which providers you need and what versions are acceptable:

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
    kubernetes = {
      source = "kubernetes/kubernetes"
      version = "~> 2.30"
    }
  }
}
```

The `source` argument identifies the provider (organization/name in the registry), and the `version` constraint specifies acceptable versions. The `~>` operator means "compatible with" – `~> 5.0` accepts any version from 5.0.0 up to but not including 6.0.0.

Provider configuration and authentication

Each provider needs to be configured before it can interact with its platform. This typically involves setting credentials, regions, projects, or other connection parameters:

```
provider "aws" {
  region = "us-east-1"
}

provider "azurerm" {
  features {}
}

provider "google" {
  project = "my-gcp-project"
  region  = "us-central1"
}
```

Credentials should never be hardcoded in configuration files. Instead, providers support various authentication methods:

- **Environment variables**: Most providers check standard environment variables first (e.g., `AWS_ACCESS_KEY_ID`, `ARM_CLIENT_SECRET`, `GOOGLE_APPLICATION_CREDENTIALS`).
- **Shared credential files**: AWS uses `~/.aws/credentials`, Azure can use the Azure CLI login, GCP can use `gcloud auth`.

- **Instance profiles and managed identities**: When running on cloud infrastructure, providers can automatically use IAM roles (AWS), Managed Identities (Azure), or Workload Identity (GCP) for credential-free authentication.
- **OIDC federation**: Modern CI/CD workflows use OIDC to obtain short-lived credentials without storing long-term secrets.

The key principle is that provider configuration lives in your OpenTofu files, but sensitive values like passwords and tokens come from the execution environment – never from version-controlled code.

Resources: The Fundamental Building Blocks

Resources are the core concept in OpenTofu. Every piece of infrastructure you manage – every virtual machine, network interface, database, DNS record, IAM policy – is represented as a resource block in your configuration.

Resource syntax

A resource block has this structure:

```
resource "TYPE" "NAME" {
  argument1 = value1
  argument2 = value2
  # ... more arguments ...
}
```

- `TYPE` is the resource type, defined by the provider (e.g., `aws_instance`, `azurerm_virtual_network`, `google_compute_disk`).
- `NAME` is a local identifier you choose to reference this resource elsewhere in your configuration.
- The body contains arguments that configure the resource – required settings and optional parameters.

Here is a concrete example creating an EC2 instance:

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  tags = {
    Name        = "web-server"
    Environment = "production"
  }
}
```

This declares that you want an EC2 instance with a specific AMI (Amazon Machine Image), a t3.micro instance type, and two tags. When you apply this configuration, the AWS provider translates it into API calls to create exactly that resource.

Resource attributes

Every resource has attributes — properties that describe its state. Some attributes are set by you in your configuration (arguments), while others are computed by the provider based on what actually exists:

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
}

# Accessing computed attributes of the instance
output "instance_public_ip" {
  value = aws_instance.web_server.public_ip
}

output "instance_arn" {
  value = aws_instance.web_server.arn
}
```

In this example, `ami` and `instance_type` are arguments you set. But `public_ip`, `arn`, and many other attributes are computed by AWS and reported back to OpenTofu — they become known only after the instance is created. You can reference these computed attributes in other resources or outputs, and OpenTofu automatically manages the dependency: any resource that uses `aws_instance.web_server.public_ip` will wait until the instance exists before it tries to use that value.

Resource lifecycle

OpenTofu manages the complete lifecycle of each resource:

- **Create**: When a resource appears in your configuration but does not exist in state, OpenTofu creates it during apply.
- **Update**: When you modify arguments of an existing resource, OpenTofu updates it in-place if the provider supports it. Some changes require replacement (destroy and recreate) because they cannot be applied to a running resource – for example, changing the AMI of an EC2 instance requires creating a new instance.
- **Delete**: When you remove a resource from your configuration, OpenTofu destroys it during apply.

OpenTofu tracks all of this in the state file, which maps each resource address (`aws_instance.web_server`) to its real-world identity (the actual EC2 instance ID) and current attribute values. This mapping is what enables accurate planning and drift detection.

Implicit dependencies

One of OpenTofu's most valuable features is automatic dependency management. When one resource references another, OpenTofu understands that a dependency exists:

```
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "public" {
  vpc_id     = aws_vpc.main.id  # Implicit dependency on aws_vpc.main
  cidr_block = "10.0.1.0/24"
}

resource "aws_instance" "web" {
  subnet_id = aws_subnet.public.id  # Implicit dependency on aws_subnet.public
  ami       = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
}
```

OpenTofu builds a dependency graph from these references and ensures resources are created in the correct order: VPC first, then subnet, then instance. It also destroys them in reverse order. You do not need to specify this ordering explicitly – it is inferred from your configuration.

When implicit dependencies are not sufficient (for example, when a resource depends on a side effect of another resource that is not expressed

through attribute references), you can use the explicit `depends_on` meta-argument:

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"

  depends_on = [aws_cloudwatch_log_group.app_logs]
}
```

Use `depends_on` sparingly and only when necessary – implicit dependencies are preferred because they make relationships explicit in your code and are easier to understand at a glance.

State: Why It Exists and How It Works

State is arguably the most important concept in OpenTofu, and also the most commonly misunderstood. Getting state management right is critical for production reliability; getting it wrong can lead to data loss, duplicate resources, or infrastructure that no longer matches your configuration.

What state actually stores

The state file (by default `terraform.tfstate`) is a JSON document that records:

1. **Resource bindings**: The mapping between OpenTofu resource addresses and real-world resource identifiers. For example, it knows that `aws_instance.web_server` corresponds to EC2 instance `i-0abc123def456`.
2. **Attribute values**: The current values of all attributes for each managed resource, as last reported by the provider.
3. **Provider configuration**: Information about which providers are in use and their configurations.
4. **Metadata**: Version information, serial number (incremented on each change), lineages for tracking state history.

Critically, state is what allows OpenTofu to answer the question: "What changes do I need to make to get from where we are now to the desired configuration?" Without state, OpenTofu would have no way of knowing what

currently exists and would have to query every API on every run – which would be slow, error-prone, and often impossible (some APIs do not support listing all resources).

How state is used in the workflow

Every time you run `tofu plan` or `tofu apply`, OpenTofu follows this sequence:

1. **Load state**: Read the current state file to understand what resources are managed.
2. **Refresh**: For each managed resource, ask the provider to read its current real-world state from the API. Update the in-memory state with fresh data. This is how drift detection works – if the API reports different values than what was stored in state, OpenTofu knows something changed outside of its control.
3. **Plan**: Compare the refreshed state against your desired configuration. Compute which resources need to be created, updated, replaced, or destroyed.
4. **Apply**: Execute the planned changes through provider APIs, updating real infrastructure.
5. **Save state**: Write the updated state back to the state file, reflecting the new reality.

This refresh-plan-apply-save cycle ensures that OpenTofu's understanding of your infrastructure is always synchronized with actual API data. The state file is not a source of truth about what should exist; your configuration is. State is simply a snapshot of what currently exists, used as input for computing changes.

Local vs. remote state

By default, OpenTofu stores state in a local file called `terraform.tfstate` in your working directory. This works fine for individual experimentation but is problematic for teams:

- **No collaboration**: Only one person can safely work with the state at a time.
- **No locking**: Concurrent operations can corrupt the state file.
- **Single point of failure**: If the local machine is lost, so is the state.

- **Security concerns**: State files often contain sensitive data (passwords, keys) and should not be stored on individual workstations or committed to version control.

For production use, you configure a remote backend that stores state in shared, secure storage with locking support:

```
terraform {
  backend "s3" {
    bucket         = "my-opentofu-state"
    key            = "production/infrastructure.tfstate"
    region         = "us-east-1"
    encrypt        = true
    use_lockfile   = true
  }
}
```

Remote backends provide:

- **Centralized storage**: All team members work with the same state.
- **State locking**: Prevents concurrent modifications that could corrupt state.
- **Versioning**: Many backends support versioned storage, enabling rollback to previous states.
- **Encryption**: State can be encrypted at rest and in transit.

Chapter 7 covers state management and backends in much greater depth, including configuration examples for all major cloud providers, migration procedures, and disaster recovery strategies.

What state does NOT store

Understanding what is not in state is equally important:

- **Configuration**: Your `.tf` files are the source of truth for desired state. State only tracks what currently exists.
- **Plan results**: Plans are computed on demand from configuration and state; they are not persisted (unless you explicitly save a plan file with `-out`).
- **Execution history**: While state has a serial number that increments on changes, it does not keep a full audit log of who made what changes and when. You need external tooling for that.

Data Sources: Reading the World Without Creating It

Data sources are OpenTofu's mechanism for reading information from existing infrastructure or external systems without creating or managing any resources. They complement managed resources by providing read-only access to data that your configuration needs as inputs.

When to use data sources

Common scenarios for data sources include:

- **Looking up existing resources**: You need the ID of an AMI, VPC, or security group that was created outside of OpenTofu and you want to reference it.
- **Dynamic configuration**: You need current information (like the latest Ubuntu AMI, or available instance types) to make decisions in your configuration.
- **Cross-project references**: One OpenTofu project needs information about resources managed by another project.

Data source syntax

Data sources use the `data` keyword instead of `resource`:

```
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
  }

  owners = ["099720109477"]  # Canonical's AWS account ID
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
}
```

This configuration finds the most recent Ubuntu 22.04 AMI published by Canonical and uses it for the EC2 instance. Every time you run `tofu plan`,

OpenTofu queries AWS to get the current latest AMI ID. The instance will automatically use the newest available image.

Data sources vs. resources

The key differences:

- **Resources are managed**: OpenTofu creates, updates, and destroys them based on your configuration.
- **Data sources are read-only**: OpenTofu only reads their current state from APIs; it never modifies or deletes the underlying objects.
- **Removing a resource** from configuration causes OpenTofu to destroy it.
- **Removing a data source** from configuration simply stops reading that information; nothing happens to the underlying infrastructure.

Data sources vs. variables

Sometimes you can achieve the same result with either a data source or a variable. The choice depends on your needs:

- Use **variables** when the value is known ahead of time and does not need to be looked up dynamically (e.g., environment name, application name).
- Use **data sources** when the value must be discovered from an external system at plan time (e.g., latest AMI ID, VPC ID from another project, list of available subnets).

Data source refresh behavior

Data sources are refreshed every time you run `tofu plan` or `tofu apply`, just like managed resources. This means:

- They always reflect current reality from the API.
- If the data they read changes (a new AMI is published, a VPC gets a new route table), your next plan may show different results.
- They add API calls to every plan operation, which can slow down planning for large configurations with many data sources.

Be judicious with data sources — use them when you genuinely need dynamic information, not as a convenience for values that could be passed as variables. Overusing data sources is a common cause of slow plan times in large OpenTofu projects.

Chapter 3: Installation and Project Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Verifying Your Installation and Checking Versions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

The Minimal Working Project

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Directory Layout Conventions: Single Project vs. Modular Organization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 4: HCL Syntax and Language Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Blocks, Arguments, Comments, and Basic Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Expressions: Interpolation, Conditionals, and Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

For Expressions: Transforming Collections Elegantly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Dynamic Blocks: Generating Repetitive Configuration Programmatically

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

Chapter 5: Variables, Locals, and Outputs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Locals: Organizing Complex Logic and Avoiding Repetition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Outputs: Exposing Information and Connecting Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Passing Data Between Modules: The Parent-Child Contract

Chapter 6: Providers, Resources, and Data Sources in Depth

How Providers Work: Plugins, Authentication, and Configuration Blocks

Resource Arguments: Required, Optional, Computed, and Deprecated

Creating Resources Across Clouds: AWS, Azure, GCP Examples

Data Sources: Querying Existing Infrastructure Safely

Provider Aliases: Managing Multiple Configurations Simultaneously

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Version Constraints: Pinning Providers for Stability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 7: State Management and Backends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Remote State Configuration: Locking, Encryption, and Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Reading Remote State: Cross-Referencing Infrastructure Between Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

State Migration: Moving Between Backends Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

State Troubleshooting: Corruption, Drift, and Recovery Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 8: Modules: Building Reusable Infrastructure Components

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

The OpenTofu Registry: Publishing and Consuming Community Modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Advanced Module Patterns: Stacks, Composition, and Versioned Releases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 9: Workspaces, Lifecycle Rules, and Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Workspaces: Multiple Environments from One Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Lifecycle Arguments: create_before_destroy, prevent_destroy, ignore_changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Dependency Management: Explicit depends_on vs. Implicit References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

The Dependency Graph: Visualizing and Debugging Execution Order

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Tainting Resources: Forcing Recreation When Needed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 10: Provisioners and Importing Existing Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

The Problem With Provisioners: Why They Are Discouraged

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Importing Existing Resources: The import Command and Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Generating Configuration for Imported Resources: State Inspection Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Migration Strategies: Bringing Legacy Infrastructure Under OpenTofu Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 11: Testing, Validation, Formatting, and Policy Enforcement

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

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

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

OpenTofu Test Framework: Built-in Testing Capabilities

Policy as Code: Sentinel, OPA, and openpolicyagent Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 12: Security Best Practices and Secrets Management

Securing State Files: Encryption at Rest and in Transit

Managing Secrets: Never Hardcoding Credentials

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

Least Privilege for OpenTofu Execution: IAM Policies and RBAC

Audit Logging and Compliance: Tracking Who Changed What and When

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Chapter 13: CI/CD Integration, GitOps, and Automation Workflows

The Infrastructure Pipeline: Commit, Plan, Review, Apply

GitHub Actions Integration: Complete Workflow Examples

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

GitOps With OpenTofu: Declarative Deployment Using ArgoCD and Flux

Approval Gates and Change Management: Human Review in Automated Workflows

Parallel Execution and Performance Optimization: Scaling Plans and Applies

Chapter 14: Enterprise-Scale Patterns and Production Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Organizational Structure: Team-Based Repository Layouts and Shared Responsibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Environment Strategy: Development, Staging, Production Separation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

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

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Debugging and Logging: Tracing Failures in Complex Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Real-World Case Studies: Production Patterns from Leading Organizations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

Conclusion: Delivering on the Infrastructure-as-Code Promise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/opentofuthedefinitivereference.