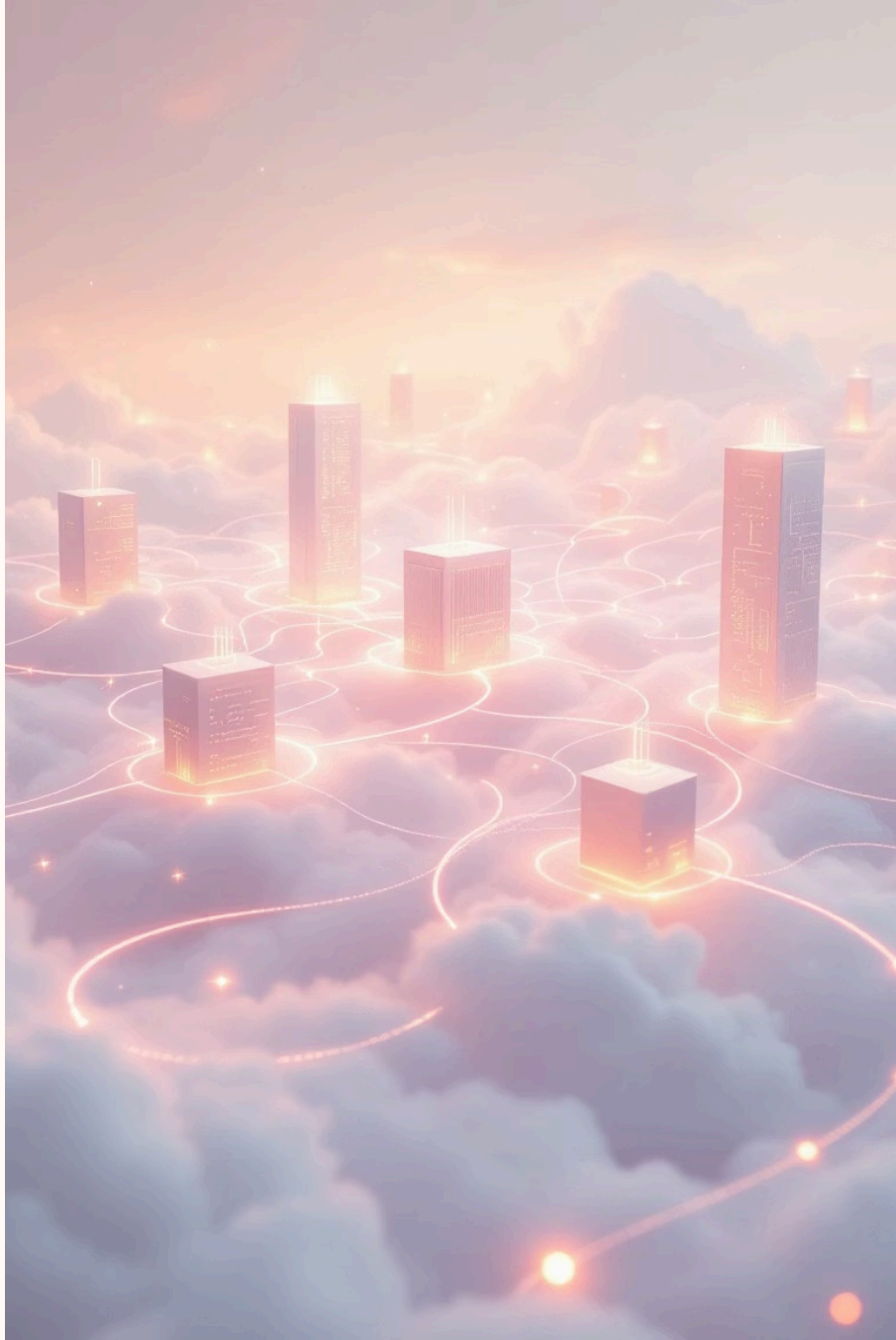
Infrastructure as Code with Terraform

A comprehensive guide to building, managing, and scaling infrastructure through code. Master the fundamentals of Terraform and modern IaC practices to transform how you provision and maintain cloud resources.



What is Infrastructure as Code?

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. This approach treats infrastructure the same way developers treat application code.

With IaC, infrastructure configurations are stored in version control systems, enabling teams to track changes, collaborate effectively, and maintain a complete history of infrastructure evolution. This methodology brings software engineering practices to infrastructure management.

Version Control

Track every infrastructure change with Git workflows

Repeatability

Deploy identical environments consistently across regions

Automation

Eliminate manual configuration and human error

Why Infrastructure as Code Matters

Speed & Efficiency

Provision complex infrastructure in minutes instead of days. Automate repetitive tasks and eliminate manual configuration bottlenecks that slow down deployment cycles.

Consistency & Reliability

Ensure every environment matches production exactly. Eliminate configuration drift and the "it works on my machine" problem through standardized deployments.

Collaboration

Enable teams to work together using familiar development workflows. Review infrastructure changes through pull requests and maintain clear documentation.

Disaster Recovery

Rebuild entire infrastructure from code in emergency scenarios. Maintain business continuity with documented, tested recovery procedures.

Terraform in the IaC Ecosystem

Terraform is an open-source infrastructure as code tool created by HashiCorp that enables you to define and provision infrastructure across multiple cloud providers using a consistent workflow. Unlike cloud-specific tools, Terraform provides a unified interface for managing resources across AWS, Azure, Google Cloud, and hundreds of other platforms.

1

Multi-Cloud Support

Manage resources across AWS, Azure, GCP, and 1000+ providers with a single tool and consistent syntax

2

Declarative Syntax

Define your desired end state and let Terraform determine the steps needed to achieve it automatically

3

Large Ecosystem

Leverage thousands of community modules and providers from the Terraform Registry

4

State Management

Track real-world infrastructure and detect drift from your defined configuration

Declarative vs. Imperative Approaches

Understanding the difference between declarative and imperative infrastructure management is crucial to mastering Terraform's philosophy and leveraging its full power.

Declarative (Terraform)

You describe WHAT you want

Define the desired end state of your infrastructure. Terraform calculates the necessary steps to reach that state, handling dependencies and ordering automatically.

```
resource "aws_instance" "web" {
  ami           = "ami-12345"
  instance_type = "t2.micro"
  count         = 3
}
```

Terraform handles creation, updates, and ordering intelligently based on your desired state.

Imperative (Scripts)

You define HOW to do it

Write explicit step-by-step instructions for every action. You must manually handle logic, error checking, and idempotency.

```
for i in 1 2 3; do
  aws ec2 run-instances \
    --image-id ami-12345 \
    --instance-type t2.micro
done
```

You're responsible for handling existing resources, dependencies, and failure scenarios.

Terraform Architecture Overview

Terraform CLI

The command-line interface you interact with to execute commands like plan, apply, and destroy. It orchestrates the entire workflow and manages the execution process.

Providers

Plugins that enable Terraform to interact with cloud platforms, SaaS providers, and APIs. Each provider exposes resources and data sources for specific platforms.

State

A record of your managed infrastructure and configuration. The state file maps your Terraform configuration to real-world resources, enabling change detection.

Modules

Reusable packages of Terraform configuration that encapsulate resource groups. Modules enable code reuse and maintain consistency across environments.

Core Terraform Concepts

Terraform's functionality is built on five fundamental concepts that work together to define, provision, and manage infrastructure. Understanding these building blocks is essential for effective infrastructure management.

1 Providers

Plugins that interface with APIs of cloud platforms and services. Providers like AWS, Azure, and Google Cloud expose resources you can manage through Terraform.

```
provider "aws" {
  region = "us-west-2"
}
```

2 Resources

The fundamental building blocks representing infrastructure components. Each resource block declares a specific infrastructure object to be created and managed.

```
resource "aws_instance" "app" {
  ami           = "ami-123"
  instance_type = "t2.micro"
}
```

3 Data Sources

Read-only queries that fetch information from providers. Use data sources to reference existing infrastructure or retrieve dynamic values during planning.

```
data "aws_ami" "latest" {
  most_recent = true
  owners      = ["amazon"]
}
```

4 Modules

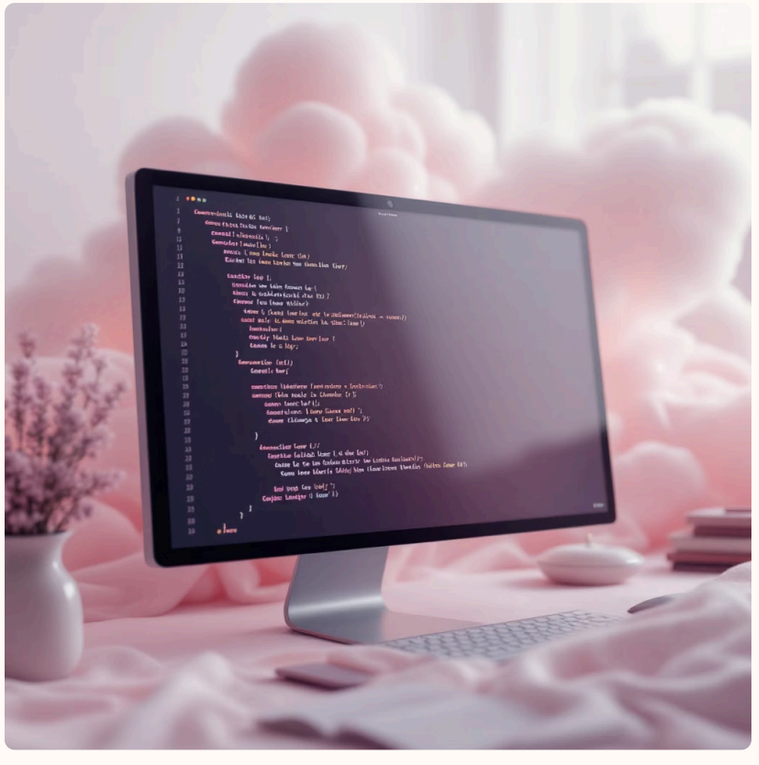
Containers for multiple resources used together as a logical unit. Modules promote reusability and help organize complex configurations into manageable components.

```
module "vpc" {
  source = "./modules/vpc"
  cidr   = "10.0.0.0/16"
}
```

5 State File

A JSON file tracking the mapping between your configuration and real infrastructure. The state enables Terraform to detect changes and plan updates accordingly.

HashiCorp Configuration Language (HCL)



HCL is Terraform's declarative configuration language designed for human readability. It uses a simple, consistent syntax based on blocks, arguments, and expressions that makes infrastructure code easy to write and understand.

Basic HCL Structure

```
# Block with type and label
resource "aws_instance" "example" {
  # Arguments (key-value pairs)
  ami           = "ami-12345"
  instance_type = "t2.micro"

  # Expressions and references
  tags = {
    Name = "web-${var.environment}"
  }
}
```

Blocks

Containers for configuration representing objects. Blocks have a type (resource, variable, output) and optional labels.

Arguments

Assign values to named parameters within blocks. Arguments use the syntax name = value.

Expressions

Compute values dynamically using references, operators, and functions. Expressions make configurations flexible and reusable.

Working with Providers and Resources

Providers and resources form the foundation of any Terraform configuration. The provider block configures access to a platform, while resource blocks define the infrastructure components you want to create.

Provider Configuration

```
provider "aws" {
  region  = "us-east-1"
  profile = "production"

  default_tags {
    tags = {
      Environment = "Production"
      ManagedBy   = "Terraform"
    }
  }
}
```

Resource Declaration

```
resource "aws_s3_bucket" "data" {
  bucket = "my-app-data-bucket"

  tags = {
    Purpose = "Application data storage"
  }
}

resource "aws_s3_bucket_versioning" "data" {
  bucket = aws_s3_bucket.data.id

  versioning_configuration {
    status = "Enabled"
  }
}
```

Resources can reference each other using the syntax resource_type.resource_name.attribute, creating implicit dependencies that Terraform uses to determine the correct order of operations.

Data Sources and the Terraform Registry

Data sources allow Terraform to fetch and compute information from external sources or existing infrastructure. The Terraform Registry serves as a central repository for providers, modules, and documentation.

Using Data Sources

```
data "aws_availability_zones" "available" {
  state = "available"
}

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

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

resource "aws_instance" "web" {
  ami               = data.aws_ami.ubuntu.id
  availability_zone = data.aws_availability_zones.available.names[0]
}
```

Terraform Registry

The Registry (registry.terraform.io) provides:

- Official and community providers for thousands of platforms
- Pre-built modules for common infrastructure patterns
- Comprehensive documentation and examples
- Version history and compatibility information

Always review module source code and check version constraints before using Registry resources in production.

Variables, Outputs, and Local Values

1 Input Variables

Parameterize configurations to make them flexible and reusable

```
variable "instance_count" {
  description = "Number of instances"
  type        = number
  default     = 2

  validation {
    condition     = var.instance_count > 0
    error_message = "Must be positive"
  }
}
```

2 Output Values

Expose information about your infrastructure after provisioning

```
output "instance_ips" {
  description = "Public IPs"
  value       = aws_instance.web[*].public_ip
  sensitive   = false
}
```

3 Local Values

Define computed values used multiple times within a module

```
locals {
  common_tags = {
    Project     = var.project_name
    Environment = var.environment
    ManagedBy   = "Terraform"
  }

  instance_name = "${var.project}-${var.environment}"
}
```

Variable Types and Files

Terraform supports various variable types including string, number, bool, list, set, map, and complex object types. Define variables in .tf files and provide values through:

- terraform.tfvars files (automatically loaded)
- *.auto.tfvars files (automatically loaded)
- Command line flags: -var="key=value"
- Environment variables: TF_VAR_name

Built-in Functions

Terraform includes 100+ built-in functions for string manipulation, numeric operations, collection functions, and more:

```
locals {
  uppercase_name = upper(var.name)
  subnet_cidrs = cidrsubnets("10.0.0.0/16", 4, 4, 4)
  config_json = jsonencode(var.config)
  merged_tags = merge(local.common_tags, var.custom_tags)
}
```

The Core Terraform Workflow

Terraform follows a consistent workflow for managing infrastructure. These core commands form the foundation of every Terraform operation, from initial setup to ongoing maintenance.

1

terraform init

Initialize a working directory containing Terraform configuration files. Downloads provider plugins, initializes the backend, and prepares modules.

2

terraform fmt

Format configuration files to a canonical style and syntax. Ensures consistency across your codebase and team.

3

terraform validate

Check configuration files for syntax errors and internal consistency without accessing remote services.

4

terraform plan

Generate and show an execution plan detailing what actions Terraform will take to reach the desired state.

5

terraform apply

Execute the actions proposed in the plan to create, update, or delete infrastructure to match your configuration.

6

terraform destroy

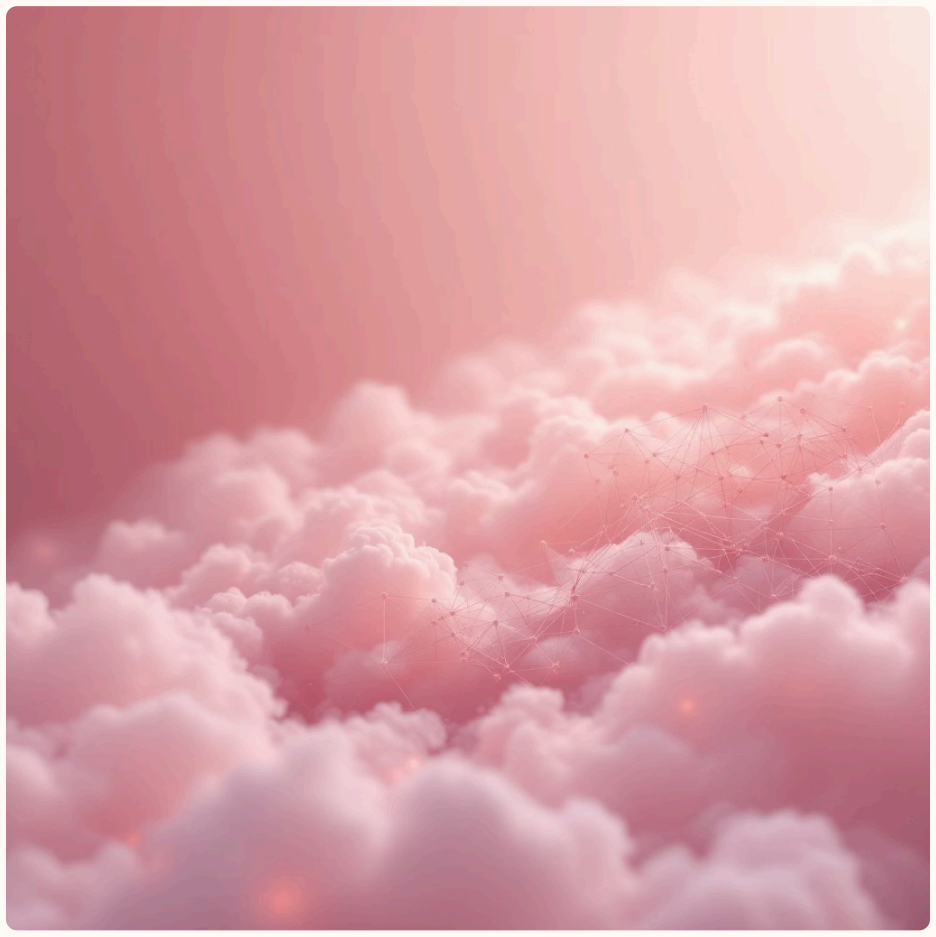
Destroy all resources managed by the current Terraform configuration. Use with extreme caution in production.

Resource Lifecycle and Dependencies

CRUD Operations

Terraform manages resources through standard CRUD (Create, Read, Update, Delete) operations. When you run terraform apply, Terraform:

- **Creates** resources that don't exist
- **Reads** current state of existing resources
- **Updates** resources when configuration changes
- **Deletes** resources removed from configuration



Dependency Management

Terraform automatically builds a dependency graph to determine the correct order for resource operations.

Implicit Dependencies

```
resource "aws_instance" "app" {
  ami           = data.aws_ami.ubuntu.id
  subnet_id     = aws_subnet.private.id
  # Terraform knows to create subnet first
}
```

Explicit Dependencies

```
resource "aws_eip" "app" {
  instance = aws_instance.app.id

  depends_on = [
    aws_internet_gateway.main
  ]
}
```

The dependency graph ensures resources are created in the correct order and can be destroyed safely by reversing that order. Use explicit depends_on only when Terraform cannot detect the dependency automatically.

Managing Desired State and Drift

Terraform maintains a desired state model where your configuration files define how infrastructure should look. The state file tracks the current reality, enabling Terraform to detect and correct drift.

1 Desired State Model

Your .tf files declare the desired end state of infrastructure. Terraform calculates the minimal set of changes needed to transform current infrastructure into the desired state, applying only necessary modifications.

2 Drift Detection

Drift occurs when infrastructure changes outside Terraform (manual console edits, automated systems, etc.). Running terraform plan detects drift by comparing the state file against actual infrastructure.

3 Lifecycle Meta-Arguments

```
resource "aws_instance" "app" {
  # ... configuration ...

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [tags]
  }
}
```

- **create_before_destroy:** Create replacement before destroying original
- **prevent_destroy:** Block resource deletion (safety measure)
- **ignore_changes:** Ignore changes to specific attributes

4 Provisioners

Execute scripts on resources during creation or destruction. **Avoid provisioners when possible** - they break Terraform's declarative model and make configurations harder to maintain. Prefer cloud-init, configuration management tools, or custom images instead.

State Management and Backends

The Terraform state file is a critical component that maps your configuration to real-world infrastructure. Proper state management is essential for team collaboration, security, and reliability.

1 Local State Basics

By default, Terraform stores state in a local terraform.tfstate file. This works for individual learning but creates serious problems for teams: no collaboration, no locking, no encryption, and risk of loss.

2 Remote Backends

Remote backends store state in shared locations like AWS S3, Google Cloud Storage, Azure Blob Storage, or Terraform Cloud. They enable team collaboration, automatic backups, and advanced features like state locking.

3 State Locking

Prevents concurrent operations that could corrupt state. Most remote backends support locking (S3 with DynamoDB, GCS, Terraform Cloud). Always enable locking for team environments.

4 State Encryption

State files contain sensitive information like passwords and private keys. Always encrypt state at rest and in transit. S3 supports encryption, Terraform Cloud encrypts automatically.

Essential State Commands

terraform state list

List all resources tracked in the state file

terraform state show

Display detailed information about a specific resource

terraform state rm

Remove resources from state without destroying them

terraform refresh

Update state file to match real-world infrastructure

Best Practice: Always use remote backends for production. Configure S3 with versioning, encryption, and DynamoDB for locking. Never commit state files to version control.