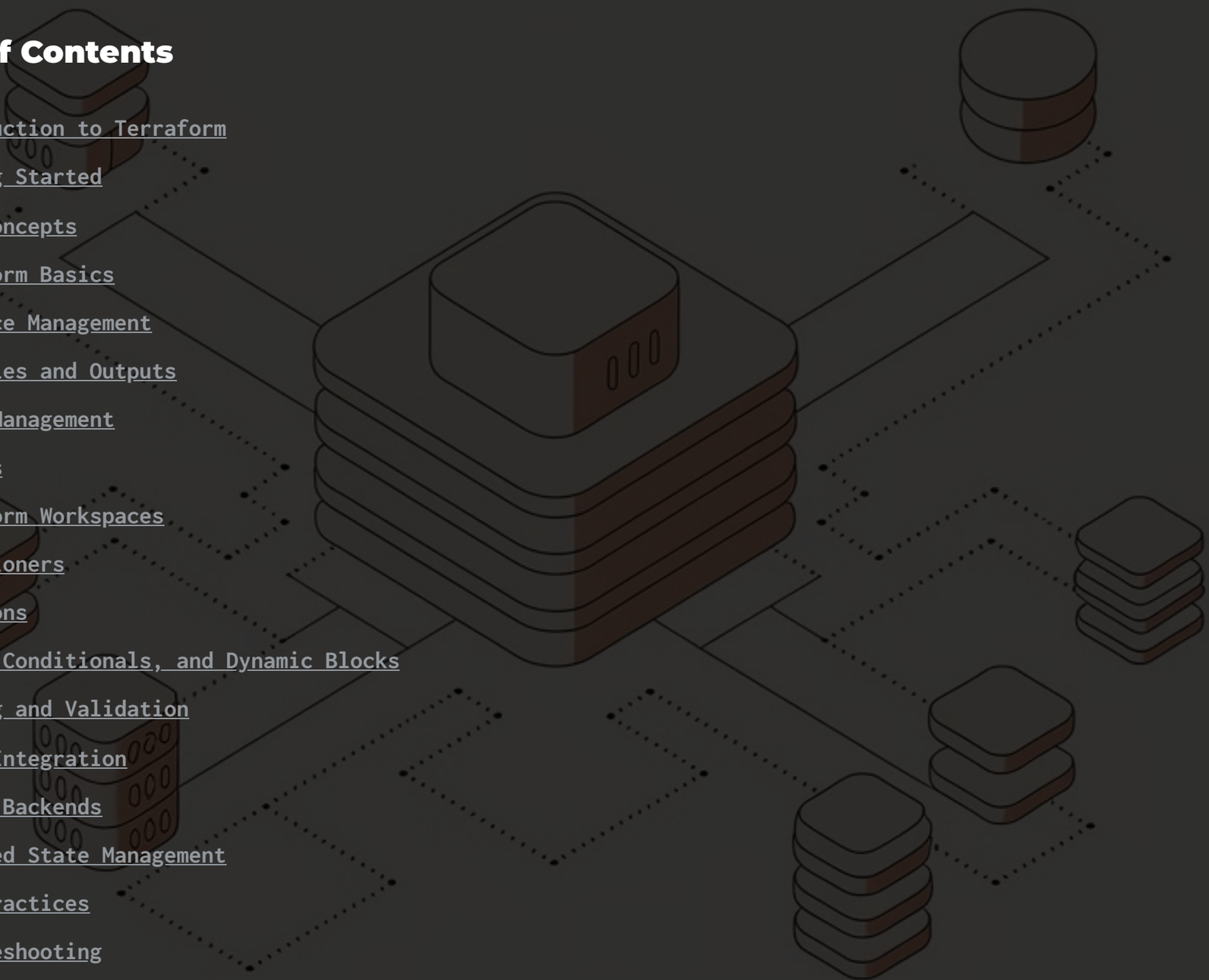
Learn Terraform in Minutes

Table of Contents



Tech Fusionist

Introduction to Terraform

What is Terraform?

Terraform is an open-source Infrastructure as Code (IaC) tool created by HashiCorp. It allows you to define and provision infrastructure using a declarative configuration language called HashiCorp Configuration Language (HCL).

Why Use Terraform?

- **Infrastructure as Code**: Define infrastructure in code files that can be versioned, reused, and shared
- **Multi-cloud support**: Works with AWS, Azure, Google Cloud, and many other providers
- **Declarative approach**: You specify the desired state, and Terraform figures out how to achieve it
- **State management**: Tracks the current state of your infrastructure
- **Dependency management**: Automatically handles resource dependencies

Key Benefits

- Consistent infrastructure deployments
- Version-controlled infrastructure
- Reduced human error
- Improved collaboration
- Faster provisioning and decommissioning

Getting Started

Installation

macOS

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

Windows

```
choco install terraform
```

Or download the binary from the **official website**.

Linux

```
# Install required packages
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl

# Add the HashiCorp GPG key (new method)
wget -O- https://apt.releases.hashicorp.com/gpg | \
gpg --dearmor | \
sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg > /dev/null

# Add the HashiCorp repository
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list

# Update and install Terraform
sudo apt-get update && sudo apt-get install terraform
```

Verify Installation

```
terraform -v
```

First Terraform Project

Create a directory for your project:

```
mkdir terraform-demo
cd terraform-demo
```

Create a file named main.tf:

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

# Create a simple AWS resource
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-example"
  }
}
```


Initialize Terraform:

```
terraform init
```

See what Terraform will do:

```
terraform plan
```

Apply the changes:

```
terraform apply
```

Clean up when done:

```
terraform destroy
```

Core Concepts

Providers

Providers are plugins that allow Terraform to interact with various cloud providers, services, and APIs.

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

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

Resources

Resources represent infrastructure objects like virtual machines, networks, etc.

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Data Sources

Data sources allow Terraform to use information defined outside of Terraform or by another Terraform configuration.

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

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  owners = ["099720109477"] # Canonical
}
```


Terraform Workflow

1. **Write** - Create or modify Terraform configuration files
2. **Initialize** - Run **terraform init** to download providers and modules
3. **Plan** - Run **terraform plan** to preview changes
4. **Apply** - Run **terraform apply** to create or update infrastructure
5. **Destroy** - Run **terraform destroy** to tear down infrastructure when no longer needed

Terraform Basics

HCL Syntax

HCL (HashiCorp Configuration Language) is the language used to write Terraform configurations.

Blocks

```
block_type "label" "name" {
  key = value
}
```

Comments

```
# This is a comment
// This is also a comment
/* This is a
   multi-line comment */
```

Strings and Interpolation

```
name = "server"
description = "This is a ${var.environment} server"
```

File Structure in Terraform

Organize your Terraform code into multiple files:

- main.tf - Main configuration
- variables.tf - Input variable declarations
- outputs.tf - Output value declarations
- terraform.tfvars - Variable assignments
- providers.tf - Provider configurations

Command Line Basics

```
# Initialize working directory
terraform init

# Format code
terraform fmt

# Validate configuration
terraform validate

# Show execution plan
terraform plan

# Apply changes
terraform apply

# Destroy resources
terraform destroy

# Show state
terraform state list
terraform state show aws_instance.example
```

Resource Management

Basic Resource Creation

```
resource "aws_s3_bucket" "data" {
  bucket = "my-data-bucket"
  acl    = "private"

  tags = {
    Environment = "Production"
    Project     = "Data Storage"
  }
}
```

Resource Dependencies

`Terraform automatically determines dependency order based on references:`

```
# Explicit dependency
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  depends_on = [aws_s3_bucket.data]
}

# Implicit dependency
resource "aws_eip" "ip" {
  instance = aws_instance.web.id
}
```

Resource Meta-Arguments

depends_on

```
resource "aws_instance" "example" {
  # ...
  depends_on = [aws_s3_bucket.example]
}
```

count

```
resource "aws_instance" "server" {
  count = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "server-${count.index}"
  }
}
```

for_each

```
resource "aws_instance" "server" {
  for_each = {
    web = "t2.micro"
    app = "t2.small"
    db  = "t2.medium"
  }

  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = each.value

  tags = {
    Name = "server-${each.key}"
  }
}
```


lifecycle

```
resource "aws_instance" "example" {
  # ...

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

provider

```
resource "aws_instance" "example" {
  provider = aws.west
  # ...
}
```

Import Existing Resources

```
terraform import aws_instance.example i-0123456789abcdef0
```

Variables and Outputs

Input Variables

Variable Declaration

```
# variables.tf
variable "region" {
  description = "AWS region to deploy resources"
  type        = string
  default     = "us-west-2"
}

variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"
}

variable "instance_count" {
  description = "Number of instances to create"
  type        = number
  default     = 1
}

variable "enabled" {
  description = "Whether to create the resources"
  type        = bool
  default     = true
}

variable "subnet_ids" {
  description = "List of subnet IDs"
  type        = list(string)
}

variable "tags" {
  description = "Tags for resources"
  type        = map(string)
  default     = {}
}

variable "instance_settings" {
  description = "Map of EC2 instance settings"
  type = object({
    ami           = string
    instance_type = string
    subnet_id     = string
    tags          = map(string)
  })
}
```

Using Variables

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "example" {
  count         = var.instance_count
  ami           = var.instance_settings.ami
  instance_type = var.instance_type
  subnet_id     = var.subnet_ids[0]

  tags = merge(var.tags, {
    Name = "example-${count.index}"
  })
}
```

Variable Assignment Methods

`In a .tfvars file:`

```
# terraform.tfvars
region       = "us-east-1"
instance_type = "t2.small"
subnet_ids    = ["subnet-12345", "subnet-67890"]
```

`Command line:`

```
terraform apply -var="region=us-east-1" -var="instance_type=t2.small"
```


`Environment variables:`

```
export TF_VAR_region=us-east-1
export TF_VAR_instance_type=t2.small
```

Local Values

`Local values can be used to simplify your configuration by avoiding repetition:`

```
locals {
  common_tags = {
    Project     = "Example"
    Environment = var.environment
    Owner       = "DevOps Team"
  }

  instance_name = "${var.environment}-instance"
}

resource "aws_instance" "example" {
  # ...
  tags = merge(local.common_tags, {
    Name = local.instance_name
  })
}
```

Output Values

`Outputs allow you to expose specific values that might be useful to the user:`

```
# outputs.tf
output "instance_id" {
  description = "ID of the EC2 instance"
  value       = aws_instance.example.id
}

output "instance_public_ip" {
  description = "Public IP address of the EC2 instance"
  value       = aws_instance.example.public_ip
}

output "instance_details" {
  description = "Map of instance details"
  value = {
    id        = aws_instance.example.id
    public_ip = aws_instance.example.public_ip
    private_ip = aws_instance.example.private_ip
    subnet_id = aws_instance.example.subnet_id
  }
  sensitive = false
}
```

State Management

Understanding Terraform State

Terraform state is a file that maps real-world resources to your configuration, tracks metadata, and improves performance.

State Files

By default, Terraform stores state locally in a file named terraform.tfstate.

State Commands

```
# List resources in state
terraform state list

# Show a specific resource
terraform state show aws_instance.example

# Move a resource to a different name
terraform state mv aws_instance.example aws_instance.web

# Remove a resource from state
terraform state rm aws_instance.old

# Import existing infrastructure
terraform import aws_instance.imported i-0123456789abcdef0

# Pull current state
terraform state pull

# Push state manually
terraform state push

# Show state file content
terraform show
```


Sensitive Data in State

State can contain sensitive data. Best practices:

- Store state remotely with proper access controls
- Enable encryption
- Use -state-out to avoid writing state to disk

Modules

What are Modules?

Modules are containers for multiple resources that are used together, allowing code reuse and organization.

Creating a Module

Structure of a basic module:

```
modules/
  ├── vpc/
  │   ├── main.tf
  │   ├── variables.tf
  │   ├── outputs.tf
  │   └── README.md
```

Example VPC module:

```
# modules/vpc/main.tf
resource "aws_vpc" "this" {
  cidr_block = var.cidr_block

  tags = merge(var.tags, {
    Name = var.name
  })
}

resource "aws_subnet" "public" {
  count = length(var.public_subnets)

  vpc_id     = aws_vpc.this.id
  cidr_block = var.public_subnets[count.index]

  tags = merge(var.tags, {
    Name = "${var.name}-public-${count.index}"
  })
}
```

```
# modules/vpc/variables.tf
variable "name" {
  description = "Name of the VPC"
  type        = string
}

variable "cidr_block" {
  description = "CIDR block for the VPC"
  type        = string
}

variable "public_subnets" {
  description = "List of public subnet CIDR blocks"
  type        = list(string)
  default     = []
}

variable "tags" {
  description = "Tags to apply to resources"
  type        = map(string)
  default     = {}
}
```

```
# modules/vpc/outputs.tf
output "vpc_id" {
  description = "ID of the VPC"
  value       = aws_vpc.this.id
}

output "public_subnet_ids" {
  description = "List of public subnet IDs"
  value       = aws_subnet.public[*].id
}
```

Using Modules

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

  name       = "example-vpc"
  cidr_block = "10.0.0.0/16"
  public_subnets = [
    "10.0.1.0/24",
    "10.0.2.0/24"
  ]

  tags = {
    Environment = "Development"
    Project     = "Example"
  }
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = module.vpc.public_subnet_ids[0]

  tags = {
    Name = "example-instance"
  }
}
```


Module Sources

Modules can be loaded from various sources:

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

  name       = "example-vpc"
  cidr_block = "10.0.0.0/16"
  public_subnets = [
    "10.0.1.0/24",
    "10.0.2.0/24"
  ]

  tags = {
    Environment = "Development"
    Project     = "Example"
  }
}

resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = module.vpc.public_subnet_ids[0]

  tags = {
    Name = "example-instance"
  }
}
```

Module Composition

Modules can be composed by using modules within modules.

Terraform Workspaces

What are Workspaces?

Workspaces allow you to manage multiple environments (like dev, staging, production) with the same configuration files but separate state.

Managing Workspaces

```
# Create a new workspace
terraform workspace new dev

# List available workspaces
terraform workspace list

# Select a workspace
terraform workspace select prod

# Show current workspace
terraform workspace show

# Delete a workspace
terraform workspace delete dev
```

Using Workspaces in Configuration

```
resource "aws_instance" "example" {
  instance_type = terraform.workspace == "prod" ? "t2.medium" : "t2.micro"

  tags = {
    Environment = terraform.workspace
    Name        = "${terraform.workspace}-instance"
  }
}
```

Provisioners

Types of Provisioners

local-exec

```
resource "aws_instance" "example" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

remote-exec

```
resource "aws_instance" "example" {
  # ...

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
      "sudo systemctl start nginx"
    ]

    connection {
      type        = "ssh"
      user        = "ubuntu"
      private_key = file("~/.ssh/id_rsa")
      host        = self.public_ip
    }
  }
}
```

file

```
resource "aws_instance" "example" {
  # ...

  provisioner "file" {
    source      = "conf/nginx.conf"
    destination = "/tmp/nginx.conf"

    connection {
      type        = "ssh"
      user        = "ubuntu"
      private_key = file("~/.ssh/id_rsa")
      host        = self.public_ip
    }
  }
}
```

Provisioner Behaviors

on-create

```
resource "aws_instance" "example" {
  # ...

  provisioner "local-exec" {
    when    = create
    command = "echo 'Instance created'"
  }
}
```


on-destroy

```
resource "aws_instance" "example" {
  # ...

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Instance destroyed'"
  }
}
```

failure behavior

```
resource "aws_instance" "example" {
  # ...

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx"
    ]

    on_failure = "continue" # or "fail"
  }
}
```

Functions

String Functions

```
locals {
  upper      = upper("hello")       # "HELLO"
  lower      = lower("WORLD")       # "world"
  title      = title("hello world")  # "Hello World"
  substr     = substr("hello", 1, 3) # "ell"
  join       = join(",", ["a", "b"]) # "a,b"
  split      = split(",", "a,b")     # ["a", "b"]
  replace    = replace("hello", "l", "L") # "heLLo"
  trim       = trim(" hello ")       # "hello"
  format     = format("Hello, %s!", "World") # "Hello, World!"
}
```

Collection Functions

```
locals {
  concat     = concat(["a"], ["b"]) # ["a", "b"]
  length     = length([1, 2, 3])    # 3
  element    = element(["a", "b"], 1) # "b"
  contains   = contains(["a", "b"], "a") # true
  keys       = keys({a = 1, b = 2})  # ["a", "b"]
  values     = values({a = 1, b = 2}) # [1, 2]
  lookup     = lookup({a = 1, b = 2}, "a", 0) # 1
  zipmap     = zipmap(["a", "b"], [1, 2]) # {a = 1, b = 2}
  merge      = merge({a = 1}, {b = 2}) # {a = 1, b = 2}
}
```

Numeric Functions

```
locals {
  abs        = abs(-42)            # 42
  ceil       = ceil(1.1)           # 2
  floor      = floor(1.9)          # 1
  max        = max(1, 2, 3)        # 3
  min        = min(1, 2, 3)        # 1
  pow        = pow(2, 3)           # 8
  signum     = signum(-42)         # -1
}
```

Date and Time Functions

```
locals {
  timestamp  = timestamp()          # "2023-01-01T12:34:56Z"
  timeadd    = timeadd(timestamp(), "1h") # Add 1 hour
  formatdate = formatdate("YYYY-MM-DD", timestamp())
}
```

IP Network Functions

```
locals {
  cidr_subnets = cidrsubnets("10.0.0.0/16", 8, 8, 8)
  # ["10.0.0.0/24", "10.0.1.0/24", "10.0.2.0/24"]

  cidr_host    = cidrhost("10.0.0.0/24", 5) # "10.0.0.5"
  cidr_netmask = cidrnetmask("10.0.0.0/24") # "255.255.255.0"
}
```


Type Conversion Functions

```
locals {
  to_string   = tostring(42)        # "42"
  to_number   = tonumber("42")      # 42
  to_bool     = tobool("true")      # true
  to_list     = tolist(["a", "b"])   # ["a", "b"]
  to_map      = tomap({a = 1, b = 2}) # {a = 1, b = 2}
  to_set      = toset(["a", "b", "a"]) # ["a", "b"]
}
```

File System Functions

```
locals {
  file_content = file("${path.module}/example.txt")
  template     = templatefile("${path.module}/template.tpl", {
    name = "John"
    items = ["apple", "banana"]
  })
}
```

Loops, Conditionals, and Dynamic Blocks

Count

```
resource "aws_instance" "server" {
  count = 3

  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "server-${count.index}"
  }
}
```

For Each

```
resource "aws_instance" "server" {
  for_each = {
    web = "t2.micro"
    app = "t2.small"
    db  = "t2.medium"
  }

  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = each.value

  tags = {
    Name = "server-${each.key}"
  }
}
```

Conditional Expressions

```
resource "aws_instance" "server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.environment == "prod" ? "t2.medium" : "t2.micro"

  ebs_block_device {
    volume_size = var.environment == "prod" ? 100 : 20
  }
}}
```

Dynamic Blocks

```
resource "aws_security_group" "example" {
  name        = "example"
  description = "Example security group"

  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.from_port
      to_port     = ingress.value.to_port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```


For Expressions

```
locals {
  instance_ids = [for inst in aws_instance.server : inst.id]

  instance_map = {
    for inst in aws_instance.server :
    inst.id => inst.public_ip
  }

  names = [for name, type in var.server_types : upper(name)]

  filtered_instances = [
    for inst in aws_instance.server :
    inst.id
    if inst.instance_type == "t2.micro"
  ]
}
```

Splat Expressions

```
locals {
  instance_ids = aws_instance.server[*].id
}
```

Testing and Validation

Variable Validation

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t2.micro"

  validation {
    condition     = contains(["t2.micro", "t2.small", "t2.medium"], var.instance_type)
    error_message = "The instance_type must be t2.micro, t2.small, or t2.medium."
  }
}
```

Custom Condition Checks

```
resource "aws_instance" "example" {
  # ...

  lifecycle {
    precondition {
      condition     = var.environment == "prod" ? var.instance_type == "t2.medium" : true
      error_message = "Production environment must use t2.medium or larger instances."
    }
  }
}

output "instance_ip" {
  value = aws_instance.example.public_ip

  postcondition {
    condition     = length(aws_instance.example.public_ip) > 0
    error_message = "The instance must have a public IP address."
  }
}
```

Terraform Built-in Validation

```
# Format and validate code
terraform fmt -check -recursive
terraform validate
```

Testing with Terratest

`Terratest is a Go library that makes it easier to write automated tests for your Terraform code:`


```
package test

import (
	"testing"
	"github.com/gruntwork-io/terratest/modules/terraform"
	"github.com/stretchr/testify/assert"
)

func TestTerraformAwsExample(t *testing.T) {
	terraformOptions := &terraform.Options{
		TerraformDir: "../examples/aws-instance",
		Vars: map[string]interface{}{
			"instance_type": "t2.micro",
		},
	}

	defer terraform.Destroy(t, terraformOptions)
	terraform.InitAndApply(t, terraformOptions)

	instanceID := terraform.Output(t, terraformOptions, "instance_id")
	assert.NotEmpty(t, instanceID)
}
```

CI/CD Integration

Terraform in GitHub Actions

```
name: Terraform

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

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v1
      with:
        terraform_version: 1.0.0

    - name: Terraform Format
      run: terraform fmt -check -recursive

    - name: Terraform Init
      run: terraform init

    - name: Terraform Validate
      run: terraform validate

    - name: Terraform Plan
      run: terraform plan -no-color
      if: github.event_name == 'pull_request'

    - name: Terraform Apply
      run: terraform apply -auto-approve
      if: github.ref == 'refs/heads/main' && github.event_name == 'push'
```

Terraform in GitLab CI

```
stages:
  - validate
  - plan
  - apply

image:
  name: hashicorp/terraform:1.0.0
  entrypoint: [""]

variables:
  TF_ROOT: ${CI_PROJECT_DIR}
  TF_STATE_NAME: default

before_script:
  - cd ${TF_ROOT}

validate:
  stage: validate
  script:
    - terraform init -backend=false
    - terraform fmt -check -recursive
    - terraform validate

plan:
  stage: plan
  script:
    - terraform init
    - terraform plan -out=tfplan
  artifacts:
    paths:
      - tfplan

apply:
  stage: apply
  script:
    - terraform init
    - terraform apply -auto-approve tfplan
  dependencies:
    - plan
  only:
    - main
```

Terraform in Jenkins Pipeline


```
pipeline {
    agent any

    tools {
        terraform 'terraform-1.0.0'
    }

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Terraform Init') {
            steps {
                sh 'terraform init'
            }
        }

        stage('Terraform Format') {
            steps {
                sh 'terraform fmt -check -recursive'
            }
        }

        stage('Terraform Validate') {
            steps {
                sh 'terraform validate'
            }
        }

        stage('Terraform Plan') {
            steps {
                sh 'terraform plan -out=tfplan'
            }
        }

        stage('Approval') {
            when {
                branch 'main'
            }
            steps {
                input message: 'Apply the terraform plan?'
            }
        }

        stage('Terraform Apply') {
            when {
                branch 'main'
            }
            steps {
                sh 'terraform apply -auto-approve tfplan'
            }
        }
    }
}
```

Remote Backends

AWS S3 Backend

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-locks"
  }
}
```

Azure Storage Backend

```
terraform {
  backend "azurerm" {
    resource_group_name  = "terraform-state-rg"
    storage_account_name = "terraformstate"
    container_name       = "terraform-state"
    key                  = "terraform.tfstate"
  }
}
```

Google Cloud Storage Backend

```
terraform {
  backend "gcs" {
    bucket = "terraform-state-bucket"
    prefix = "terraform/state"
  }
}
```


Terraform Cloud Backend

```
terraform {
  backend "remote" {
    organization = "example-org
```

Advanced State Management

Beyond basic state operations for complex Terraform deployments

1

State Locking

Prevents concurrent state modifications and conflicts

2

State Import

Bring existing infrastructure under Terraform management

3

Partial State Operations

Targeted apply, refresh, and state manipulation

4

State Migration

Move resources between state files or workspaces

```
# Import existing resources
terraform import aws_instance.web i-abcd1234

# State manipulation commands
terraform state list
terraform state show aws_instance.web
terraform state mv aws_s3_bucket.old aws_s3_bucket.new
terraform state rm aws_instance.old
```

Terraform Best Practices

Follow these proven patterns to create maintainable, secure infrastructure code

1

Code Organization

Structure repos by environment and component. Use consistent naming conventions for all resources.

2

Version Control

Pin provider versions and modules. Store state remotely with proper locking mechanisms.

3

Security First

Restrict IAM permissions. Never hardcode secrets. Use environment variables or secure vaults.

4

Validation Pipeline

Implement terraform fmt, validate, and plan in CI/CD. Add automated testing with Terratest.

Terraform Troubleshooting

1 Initialization Issues

Run terraform init with -upgrade flag to refresh providers and modules.

Error: Failed to query available provider packages
Solution: terraform init -upgrade

2 Plan and Apply Failures

Use -target flag to isolate problematic resources during troubleshooting.

Error: Error creating EC2 instance
Solution: terraform apply -target=aws_vpc.main

3 State Conflicts

Resolve state locks with force-unlock when processes terminate unexpectedly.

Error: Error acquiring the state lock
Solution: terraform force-unlock LOCK_ID

4 Provider Authentication

Set TF_LOG=DEBUG to see detailed API calls for credential issues.

5 Version Incompatibility

Specify provider versions explicitly to avoid breaking changes.

Error: No valid credential sources found
Solution: export TF_LOG=DEBUG

Error: Provider produced incompatible API result
Solution: version = "~> 3.0" in provider block

6 Module Source Issues

Check network connectivity when modules fail to download.

Error: Failed to download module
Solution: Check firewall settings or use local modules

7 Resource Dependencies

Use depends_on attribute to ensure proper resource creation order.

Error: Resource not found during apply
Solution: Add depends_on = [aws_vpc.main] to resources

8 Variable Validation

Add validation blocks to variables to catch errors early.

Error: Invalid variable value
Solution: Add validation block with condition and error_message

9 Remote Backend Failures

Check access permissions when remote state operations fail.

Error: Failed to load state from S3
Solution: Verify IAM permissions for S3 bucket access

10 Output Issues

Use terraform refresh when outputs don't match actual infrastructure.

11 Workspace Problems

Verify current workspace when resources appear missing.

Error: Incorrect or missing output values
Solution: terraform refresh followed by terraform output

Error: Expected resources not found
Solution: terraform workspace select CORRECT_WORKSPACE

12 Resource Quotas

Check service quotas when resource creation fails repeatedly.

13 State Corruption

Restore from backup when state file becomes corrupted.

Error: API rate limit exceeded
Solution: Implement wait time or request quota increase

Error: Failed to decode state file
Solution: Use terraform state pull > backup.tfstate before fixes

14 Terraform Crashes

Increase memory when Terraform crashes during complex operations.

Error: Segmentation fault or unexpected crash
Solution: export TF_PLUGIN_CACHE_DIR="/tmp/terraform-cache"