Infrastructure as Code with Terraform

Comprehensive Guide to Modern Infrastructure Management

Prepared By: Rashi Rana
Corporate Trainer

Why Infrastructure as Code (IaC)?

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.

Traditional Infrastructure Challenges

- **Manual Configuration:** Time-consuming, error-prone manual server setup and configuration
- **Configuration Drift:** Servers diverge from their intended configuration over time
- **Lack of Version Control:** No tracking of infrastructure changes or rollback capabilities
- **Inconsistent Environments:** Differences between development, staging, and production
- **Documentation Issues:** Infrastructure setup knowledge trapped in individual minds
- **Scaling Difficulties:** Manual processes don't scale with growing infrastructure needs

IaC Benefits

Consistency & Reliability

Version Control

Infrastructure changes tracked in Git with full

Identical infrastructure across all environments, eliminating "works on my machine" issues.

history, branching, and collaboration features.

Automation & Speed

Rapid provisioning and deployment of complex infrastructure in minutes, not hours or days.

Cost Optimization

Programmatic resource management enables automatic scaling and cost-effective resource allocation.

IaC in DevOps Culture

IaC enables the DevOps principle of treating infrastructure the same way as application code, with testing, code reviews, and continuous integration/deployment practices.

Prepared By: Rashi Rana
Corporate Trainer

Terraform Introduction

> **HashiCorp Terraform** is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.

What Makes Terraform Special?

- **Multi-Cloud Support:** Works with AWS, Azure, GCP, and 1000+ other providers
- **Declarative Language:** Describe desired end state, not step-by-step instructions
- **State Management:** Tracks real-world resources and their configuration
- **Plan Before Apply:** Preview changes before making them
- **Resource Graph:** Understands dependencies and creates resources in correct order
- **Immutable Infrastructure:** Replace rather than modify existing resources

Terraform vs Other IaC Tools

Tool	Approach	Cloud Support	Language
Terraform	Declarative	Multi-cloud	HCL (HashiCorp Configuration Language)
CloudFormation	Declarative	AWS only	JSON/YAML
Ansible	Procedural	Multi-cloud	YAML
Pulumi	Declarative	Multi-cloud	Python, TypeScript, Go, C#

Terraform Architecture

Terraform uses a plugin-based architecture where providers implement resource types for specific platforms. The core Terraform engine handles state management, planning, and execution.

Prepared By: Rashi Rana
Corporate Trainer

Terraform Providers

What are Providers?

> **Providers** are plugins that Terraform uses to interact with cloud platforms, SaaS providers, and other APIs. Each provider adds a set of resource types and data sources that Terraform can manage.

Provider Configuration

```
# Configure the AWS Provider
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "us-west-2"
  # Optional: profile = "default"
  # Optional: access_key and secret_key
}
```

Popular Providers

AWS Provider

Azure Provider

Manages AWS resources like EC2, S3, VPC, RDS, Lambda, and 400+ other services.

Manages Microsoft Azure resources including VMs, storage accounts, and networking.

Google Cloud Provider

Manages GCP resources like Compute Engine, Cloud Storage, and Kubernetes Engine.

Kubernetes Provider

Manages Kubernetes resources like deployments, services, and config maps.

Provider Authentication

- **Environment Variables:** AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY
- **Shared Credentials File:** ~/.aws/credentials
- **IAM Roles:** For EC2 instances or ECS tasks
- **Provider Configuration:** Direct specification in provider block

Security Best Practice: Never hardcode credentials in Terraform files. Use environment variables, IAM roles, or credential files instead.

Terraform Resources

Understanding Resources

> **Resources** are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components like DNS records.

Resource Syntax

```
resource "resource_type" "resource_name" {
  # Configuration arguments
  argument1 = value1
  argument2 = value2
}

# Example: AWS EC2 Instance
resource "aws_instance" "web_server" {
  ami = "ami-0c02fb55956c7d316"
  instance_type = "t3.micro"
  key_name = "my-key-pair"

  tags = {
    Name = "WebServer"
    Environment = "Production"
  }
}
```

Common AWS Resources

Resource Type	Purpose	Example Usage
`aws_instance`	EC2 virtual machines	Web servers, application servers
`aws_s3_bucket`	Object storage	Static websites, data storage
`aws_vpc`	Virtual private cloud	Network isolation
`aws_security_group`	Firewall rules	Network access control
`aws_lb`	Load balancer	Traffic distribution

Resource Dependencies

Terraform automatically determines resource dependencies based on references between resources. You can also specify explicit dependencies using the `depends_on` argument.

```
resource "aws_security_group" "web_sg" {
  name_prefix = "web-sg"
  vpc_id = aws_vpc.main.id
}

resource "aws_instance" "web" {
  ami = "ami-0c02fb55956c7d316"
  instance_type = "t3.micro"
  vpc_security_group_ids = [aws_security_group.web_sg.id]
  # Implicit dependency on aws_security_group.web_sg
}
```

Prepared By: Rashi Rana
Corporate Trainer

Terraform Variables

Input Variables

> **Input variables** serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code.

Variable Declaration

```
# variables.tf
variable "instance_type" {
  description = "EC2 instance type"
  type = string
  default = "t3.micro"
}

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

variable "availability_zones" {
  description = "List of availability zones"
  type = list(string)
  default = ["us-west-2a", "us-west-2b"]
}
```

Variable Types

Primitive Types

`string`, `number`, `bool`

Collection Types

`list(type)`, `set(type)`, `map(type)`

Structural Types

`object({...})`, `tuple([...])`

Special Type

any - accepts any type

Using Variables

```
# main.tf
resource "aws_instance" "web" {
  ami = "ami-0c02fb55956c7d316"
  instance_type = var.instance_type

  tags = {
    Name = "${var.environment}-web-server"
    Environment = var.environment
  }
}
```

Providing Variable Values

- **Command Line:** `terraform apply -var="instance_type=t3.small"`
- **Variable Files:** `terraform.tfvars` or `*.auto.tfvars`
- **Environment Variables:** `TF_VAR_instance_type=t3.small`

- **Interactive Input:** Terraform prompts for missing variables

Prepared By: Rashi Rana
Corporate Trainer

Terraform Outputs

Understanding Outputs

> **Output values** make information about your infrastructure available on the command line and can expose information for other Terraform configurations to use.

Output Declaration

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

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

output "vpc_id" {
  description = "ID of the VPC"
  value = aws_vpc.main.id
  sensitive = false
}

output "database_password" {
  description = "Database administrator password"
  value = aws_db_instance.db.password
  sensitive = true
}
```

Output Features

- **Description:** Human-readable description of the output value
- **Sensitive:** Mark outputs containing sensitive information
- **Depends_on:** Explicit dependencies for outputs
- **Precondition:** Validation rules for output values

Using Outputs

Command Line

`terraform output` `terraform output instance_id`

Module Composition

Child module outputs become available to parent module

Remote State

Access outputs from other Terraform configurations

CI/CD Integration

Pass infrastructure details to deployment pipelines

Complex Output Examples

```
output "load_balancer_info" {
  description = "Load balancer information"
  value = {
    dns_name = aws_lb.main.dns_name
    zone_id = aws_lb.main.zone_id
    arn = aws_lb.main.arn
  }
}

output "instance_ips" {
```

```
  description = "List of instance IP addresses"
  value = aws_instance.web[*].private_ip
}
```

Prepared By: Rashi Rana
Corporate Trainer

Terraform Workflow

The Core Terraform Workflow

> Terraform has a simple workflow consisting of four main commands that you'll use for most operations: **init**, **plan**, **apply**, and **destroy**.

Workflow Overview

terraform init → **terraform plan** →
terraform apply → **terraform destroy**

Command Purposes

Command	Purpose	When to Use
terraform init	Initialize working directory	First time setup, provider updates
terraform plan	Preview changes	Before applying changes, code review
terraform apply	Create/update infrastructure	Deploy changes to infrastructure

Command	Purpose	When to Use
terraform destroy	Remove all resources	Clean up, environment teardown

Additional Useful Commands

- **terraform validate:** Check configuration syntax
- **terraform fmt:** Format configuration files
- **terraform show:** Display current state
- **terraform output:** Display output values
- **terraform state:** Advanced state management

> **Best Practice:** Always run `terraform plan` before `terraform apply` to review changes and avoid unexpected modifications.

Prepared By: Rashi Rana
Corporate Trainer

terraform init

Initialization Process

> **terraform init** initializes a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration.

What terraform init Does

- **Downloads Providers:** Installs required provider plugins
- **Initializes Backend:** Configures state storage backend
- **Downloads Modules:** Installs child modules referenced in configuration
- **Creates Lock File:** Generates .terraform.lock.hcl for version consistency

Command Examples

```
# Basic initialization
terraform init

# Initialize with backend configuration
terraform init -backend-config="bucket=my-terraform-
state"

# Upgrade providers to latest versions
terraform init -upgrade

# Reconfigure backend (migrate state)
terraform init -migrate-state
```

```
# Initialize without downloading plugins (CI/CD)
terraform init -plugin-dir=/path/to/plugins
```

Directory Structure After Init

```
my-terraform-project/
├── main.tf
├── variables.tf
├── outputs.tf
├── .terraform/ # Created by init
│   ├── providers/ # Downloaded providers
│   └── modules/ # Downloaded modules
├── .terraform.lock.hcl # Provider version lock
└── terraform.tfstate # State file (after apply)
```

Common Init Scenarios

New Project

Run `terraform init` in directory with .tf files

Provider Updates

Use `terraform init -upgrade` to update providers

Backend Changes

Reconfigure with `terraform init -reconfigure`

Team Collaboration

Commit .terraform.lock.hcl for consistent provider versions

Important: The .terraform directory should not be committed to version control. Add it to your .gitignore file.

Prepared By: Rashi Rana
Corporate Trainer

terraform plan

Planning Infrastructure Changes

> **terraform plan** creates an execution plan, showing what actions Terraform will take to reach the desired state defined in your configuration files.

Plan Output Symbols

Symbol	Action	Description
+	Create	Resource will be created
-	Destroy	Resource will be destroyed
~	Update	Resource will be modified in-place
-/+	Replace	Resource will be destroyed and recreated
<=	Read	Data source will be read

Plan Command Options

```
# Basic plan
terraform plan

# Save plan to file
terraform plan -out=tfplan
```

```
# Plan with variable values
terraform plan -var="instance_type=t3.small"

# Plan with variable file
terraform plan -var-file="production.tfvars"

# Target specific resources
terraform plan -target=aws_instance.web

# Detailed exit codes
terraform plan -detailed-exitcode
```

Example Plan Output

```
Terraform will perform the following actions:

  # aws_instance.web will be created
  + resource "aws_instance" "web" {
      + ami = "ami-0c02fb55956c7d316"
      + instance_type = "t3.micro"
      + id = (known after apply)
      + public_ip = (known after apply)
      + tags = {
          + "Name" = "WebServer"
        }
    }

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

Plan Best Practices

- **Always Review:** Carefully examine plan output before applying
- **Save Plans:** Use `-out` flag for consistent apply operations
- **CI/CD Integration:** Automate plan generation in pull requests
- **Resource Targeting:** Use `-target` for selective planning

terraform apply

Applying Infrastructure Changes

> **terraform apply** executes the actions proposed in a Terraform plan to create, update, or destroy infrastructure resources.

Apply Process

1. **Generate Plan:** If no saved plan provided, generates execution plan
2. **Show Plan:** Displays planned changes for review
3. **Confirmation:** Prompts for approval (unless auto-approved)
4. **Execute Changes:** Applies changes in dependency order
5. **Update State:** Records current infrastructure state

Apply Command Options

```
# Interactive apply (prompts for confirmation)
terraform apply

# Auto-approve (no confirmation prompt)
terraform apply -auto-approve

# Apply saved plan
terraform apply tfplan

# Apply with variables
terraform apply -var="environment=production"
```

```
# Target specific resources
terraform apply -target=aws_instance.web

# Parallel resource creation (default: 10)
terraform apply -parallelism=5
```

Apply Output Example

```
aws_instance.web: Creating...
aws_instance.web: Still creating... [10s elapsed]
aws_instance.web: Still creating... [20s elapsed]
aws_instance.web: Creation complete after 22s [id=i-
0123456789abcdef0]

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

Outputs:

instance_id = "i-0123456789abcdef0"
instance_public_ip = "54.123.45.67"
```

State Management

- **State File:** terraform.tfstate tracks resource mappings
- **State Locking:** Prevents concurrent modifications
- **Remote State:** Store state in S3, Consul, or Terraform Cloud
- **State Backup:** Automatic backup before modifications

> **Important:** Never manually edit the state file. Use `terraform state` commands for state modifications.

terraform destroy

Destroying Infrastructure

> **terraform destroy** is used to destroy all resources managed by a Terraform configuration. This command is the inverse of `terraform apply`.

Destroy Process

1. **Generate Destroy Plan:** Creates plan to destroy all resources
2. **Show Destruction Plan:** Lists resources to be destroyed
3. **Confirmation:** Prompts for approval with resource count
4. **Execute Destruction:** Destroys resources in reverse dependency order
5. **Clean State:** Removes destroyed resources from state

Destroy Command Options

```
# Interactive destroy (prompts for confirmation)
terraform destroy

# Auto-approve destruction
terraform destroy -auto-approve

# Destroy specific resources
terraform destroy -target=aws_instance.web

# Destroy with variables
```

```
terraform destroy -var-file="production.tfvars"

# Plan destruction only (don't execute)
terraform plan -destroy
```

Destroy Output Example

```
aws_instance.web: Refreshing state... [id=i-
0123456789abcdef0]

Terraform will perform the following actions:

  # aws_instance.web will be destroyed
  - resource "aws_instance" "web" {
      - ami = "ami-0c02fb55956c7d316" -> null
      - instance_type = "t3.micro" -> null
      - id = "i-0123456789abcdef0" -> null
    }

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

Do you really want to destroy all resources?
  Enter a value: yes

aws_instance.web: Destroying... [id=i-0123456789abcdef0]
aws_instance.web: Still destroying... [id=i-
0123456789abcdef0, 10s elapsed]
aws_instance.web: Destruction complete after 15s

Destroy complete! Resources: 1 destroyed.
```

Destroy Safety Considerations

Data Loss Prevention

Use `prevent_destroy` lifecycle rule for critical

Selective Destruction

Use `-target` to destroy specific resources only resources

Backup Strategy

Backup data before destroying databases or storage

Environment Isolation

Use separate state files for different environments

Caution: `terraform destroy` will permanently delete all managed resources. Always verify the destruction plan before confirming.

Prepared By: Rashi Rana
Corporate Trainer

Lab: Basic Terraform Code for AWS

Setting Up Your First Terraform Project

Project Structure

```
# Create project directory
mkdir terraform-aws-lab
cd terraform-aws-lab

# Create configuration files
touch main.tf variables.tf outputs.tf
terraform.tfvars
```

1. Provider Configuration (main.tf)

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

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

2. Variables Definition (variables.tf)

```
variable "aws_region" {
  description = "AWS region"
  type = string
  default = "us-west-2"
}

variable "environment" {
  description = "Environment name"
  type = string
  default = "dev"
}

variable "instance_type" {
  description = "EC2 instance type"
  type = string
  default = "t3.micro"
}
```

3. Variable Values (terraform.tfvars)

```
aws_region = "us-west-2"
environment = "development"
instance_type = "t3.micro"
```

Prepared By: Rashi Rana
Corporate Trainer

Lab: EC2 and S3 Resources

Creating EC2 Instance and S3 Bucket

EC2 Instance Configuration

```
# Data source for latest Amazon Linux AMI
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners = ["amazon"]

  filter {
    name = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

# EC2 Instance
resource "aws_instance" "web_server" {
  ami = data.aws_ami.amazon_linux.id
  instance_type = var.instance_type

  tags = {
    Name = "${var.environment}-web-server"
    Environment = var.environment
    Project = "terraform-lab"
  }
}
```

S3 Bucket Configuration

```
# S3 Bucket
resource "aws_s3_bucket" "app_bucket" {
  bucket = "${var.environment}-terraform-lab-
bucket-${random_id.bucket_suffix.hex}"
```

```
  tags = {
    Name = "${var.environment}-app-bucket"
    Environment = var.environment
    Project = "terraform-lab"
  }
}

# Random ID for unique bucket naming
resource "random_id" "bucket_suffix" {
  byte_length = 4
}

# S3 Bucket versioning
resource "aws_s3_bucket_versioning"
"app_bucket_versioning" {
  bucket = aws_s3_bucket.app_bucket.id
  versioning_configuration {
    status = "Enabled"
  }
}
```

Outputs Configuration (outputs.tf)

```
output "instance_id" {
  description = "ID of the EC2 instance"
  value = aws_instance.web_server.id
}

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

output "s3_bucket_name" {
  description = "Name of the S3 bucket"
  value = aws_s3_bucket.app_bucket.bucket
}

output "s3_bucket_arn" {
  description = "ARN of the S3 bucket"
```

```
  value = aws_s3_bucket.app_bucket.arn
}
```

Prepared By: Rashi Rana
Corporate Trainer

Lab: Terraform Commands Execution

Step-by-Step Execution

1. **Initialize Terraform:**

```
# Initialize the working directory
terraform init

# Expected output:
# - Downloads AWS provider
# - Creates .terraform directory
# - Creates .terraform.lock.hcl
```

2. **Validate Configuration:**

```
# Validate syntax and configuration
terraform validate

# Format configuration files
terraform fmt
```

3. **Plan Infrastructure:**

```
# Create execution plan
terraform plan

# Save plan to file
terraform plan -out=tfplan
```

```
# Expected: Shows 3 resources to add (EC2,
S3, Random ID)
```

4 **Apply Changes:**

```
# Apply the configuration
terraform apply

# Or apply saved plan
terraform apply tfplan

# Auto-approve for automation
terraform apply -auto-approve
```

5 **Verify Resources:**

```
# Show current state
terraform show

# Display outputs
terraform output

# Get specific output
terraform output instance_id
```

Lab: VPC and Networking Resources

Creating VPC Infrastructure

VPC Configuration

```
# VPC
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support = true

  tags = {
    Name = "${var.environment}-vpc"
    Environment = var.environment
  }
}

# Internet Gateway
resource "aws_internet_gateway" "main" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${var.environment}-igw"
    Environment = var.environment
  }
}
```

Subnets Configuration

```
# Public Subnet
resource "aws_subnet" "public" {
```

```
  count = 2
  vpc_id = aws_vpc.main.id
  cidr_block = "10.0.${count.index + 1}.0/24"
  availability_zone =
data.aws_availability_zones.available.names[count.index
  map_public_ip_on_launch = true

  tags = {
    Name = "${var.environment}-public-
subnet-${count.index + 1}"
    Environment = var.environment
    Type = "Public"
  }
}

# Private Subnet
resource "aws_subnet" "private" {
  count = 2
  vpc_id = aws_vpc.main.id
  cidr_block = "10.0.${count.index + 10}.0/24"
  availability_zone =
data.aws_availability_zones.available.names[count.index

  tags = {
    Name = "${var.environment}-private-
subnet-${count.index + 1}"
    Environment = var.environment
    Type = "Private"
  }
}
```

Route Tables

```
# Public Route Table
resource "aws_route_table" "public" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.main.id
  }

  tags = {
```

```
    Name = "${var.environment}-public-rt"
    Environment = var.environment
  }
}

# Route Table Association
resource "aws_route_table_association" "public" {
  count = length(aws_subnet.public)
  subnet_id = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}
```

Prepared By: Rashi Rana
Corporate Trainer

Lab: Security Groups and Load Balancer

Security and Load Balancing

Security Groups

```
# Web Server Security Group
resource "aws_security_group" "web" {
  name_prefix = "${var.environment}-web-sg"
  vpc_id = aws_vpc.main.id

  ingress {
    description = "HTTP"
    from_port = 80
    to_port = 80
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "HTTPS"
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "SSH"
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["10.0.0.0/16"]
  }
```

```
  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.environment}-web-sg"
    Environment = var.environment
  }
}
```

Application Load Balancer

```
# Application Load Balancer
resource "aws_lb" "main" {
  name = "${var.environment}-alb"
  internal = false
  load_balancer_type = "application"
  security_groups = [aws_security_group.alb.id]
  subnets = aws_subnet.public[*].id

  enable_deletion_protection = false

  tags = {
    Name = "${var.environment}-alb"
    Environment = var.environment
  }
}

# ALB Target Group
resource "aws_lb_target_group" "web" {
  name = "${var.environment}-web-tg"
  port = 80
  protocol = "HTTP"
  vpc_id = aws_vpc.main.id

  health_check {
    enabled = true
    healthy_threshold = 2
    interval = 30
```

```
    matcher = "200"
    path = "/"
    port = "traffic-port"
    protocol = "HTTP"
    timeout = 5
    unhealthy_threshold = 2
  }

  tags = {
    Name = "${var.environment}-web-tg"
    Environment = var.environment
  }
}
```

ALB Listener

```
# ALB Listener
resource "aws_lb_listener" "web" {
  load_balancer_arn = aws_lb.main.arn
  port = "80"
  protocol = "HTTP"

  default_action {
    type = "forward"
    target_group_arn = aws_lb_target_group.web.arn
  }
}
```

Advanced Terraform: Modules

What are Terraform Modules?

> **Modules** are containers for multiple resources that are used together. A module consists of a collection of .tf files kept together in a directory.

Module Benefits

- **Reusability:** Write once, use multiple times across projects
- **Organization:** Logical grouping of related resources
- **Encapsulation:** Hide complexity behind simple interfaces
- **Versioning:** Version control for infrastructure components
- **Testing:** Test infrastructure components in isolation

Module Structure

```
modules/
└── vpc/
    ├── main.tf # Resources
    ├── variables.tf # Input variables
    ├── outputs.tf # Output values
    └── README.md # Documentation
```

Creating a VPC Module

```
# modules/vpc/variables.tf
variable "vpc_cidr" {
  description = "CIDR block for VPC"
  type = string
  default = "10.0.0.0/16"
}

variable "environment" {
  description = "Environment name"
  type = string
}

variable "availability_zones" {
  description = "List of availability zones"
  type = list(string)
}
```

Using Modules

```
# main.tf
module "vpc" {
  source = "./modules/vpc"

  vpc_cidr = "10.0.0.0/16"
  environment = "production"
  availability_zones = ["us-west-2a", "us-west-2b"]
}

# Reference module outputs
resource "aws_instance" "web" {
  ami = "ami-0c02fb55956c7d316"
  instance_type = "t3.micro"
  subnet_id = module.vpc.public_subnet_ids[0]
}
```

Module Sources

- **Local Paths:** `source = "./modules/vpc"`

- **Git Repositories:** `source = "git::https://github.com/user/repo.git"`
- **Terraform Registry:** `source = "terraform-aws-modules/vpc/aws"`
- **HTTP URLs:** `source = "https://example.com/module.zip"`

Prepared By: Rashi Rana
Corporate Trainer

State Files and Remote Backend

Terraform State

> **Terraform state** is used to map real world resources to your configuration, keep track of metadata, and improve performance for large infrastructures.

State File Challenges

- **Team Collaboration:** Multiple developers need access to same state
- **State Locking:** Prevent concurrent modifications
- **Security:** State files may contain sensitive information
- **Backup:** Prevent state file loss
- **Versioning:** Track state changes over time

Remote Backend Configuration

```
# backend.tf
terraform {
  backend "s3" {
    bucket = "my-terraform-state-bucket"
    key = "prod/terraform.tfstate"
    region = "us-west-2"
    encrypt = true
    dynamodb_table = "terraform-state-lock"
  }
}
```

S3 Backend Setup

```
# Create S3 bucket for state
resource "aws_s3_bucket" "terraform_state" {
  bucket = "my-terraform-state-bucket"
}

resource "aws_s3_bucket_versioning" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource
"aws_s3_bucket_server_side_encryption_configuration"
"terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id

  rule {
    apply_server_side_encryption_by_default {
      sse_algorithm = "AES256"
    }
  }
}

# DynamoDB table for state locking
resource "aws_dynamodb_table" "terraform_state_lock" {
  name = "terraform-state-lock"
  billing_mode = "PAY_PER_REQUEST"
  hash_key = "LockID"

  attribute {
    name = "LockID"
    type = "S"
  }
}
```

State Management Commands

```
# List resources in state
terraform state list

# Show specific resource
terraform state show aws_instance.web

# Move resource in state
terraform state mv aws_instance.old aws_instance.new

# Remove resource from state
terraform state rm aws_instance.web

# Import existing resource
terraform import aws_instance.web i-1234567890abcdef0
```

Prepared By: Rashi Rana
Corporate Trainer

Data Sources and Provisioners

Data Sources

> **Data sources** allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

Common Data Sources

```
# Get latest Amazon Linux AMI
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners = ["amazon"]

  filter {
    name = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

# Get availability zones
data "aws_availability_zones" "available" {
  state = "available"
}

# Get current AWS caller identity
data "aws_caller_identity" "current" {}

# Get existing VPC
data "aws_vpc" "existing" {
  filter {
    name = "tag:Name"
    values = ["production-vpc"]
```

```
  }
}
```

Using Data Sources

```
resource "aws_instance" "web" {
  ami = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"
  availability_zone =
data.aws_availability_zones.available.names[0]

  tags = {
    Name = "web-server"
    Owner = data.aws_caller_identity.current.user_id
  }
}
```

Provisioners

> **Provisioners** can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

File and Remote-exec Provisioners

```
resource "aws_instance" "web" {
  ami = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"
  key_name = "my-key"

  # File provisioner
  provisioner "file" {
    source = "app.conf"
    destination = "/tmp/app.conf"

    connection {
```

```
      type = "ssh"
      user = "ec2-user"
      private_key = file("~/.ssh/id_rsa")
      host = self.public_ip
    }
  }

  # Remote-exec provisioner
  provisioner "remote-exec" {
    inline = [
      "sudo yum update -y",
      "sudo yum install -y httpd",
      "sudo systemctl start httpd",
      "sudo systemctl enable httpd"
    ]

    connection {
      type = "ssh"
      user = "ec2-user"
      private_key = file("~/.ssh/id_rsa")
      host = self.public_ip
    }
  }
}
```

Best Practice: Use provisioners as a last resort. Consider using user_data, cloud-init, or configuration management tools like Ansible instead.

Hands-on: 2-Tier Architecture (Web + DB)

Architecture Overview

> We'll deploy a complete 2-tier architecture with web servers in public subnets and a database in private subnets, including load balancing and security groups.

Architecture Components

Network Layer

VPC, Public/Private Subnets, Internet Gateway, NAT Gateway, Route Tables

Web Tier

Application Load Balancer, Auto Scaling Group, EC2 Instances in Public Subnets

Database Tier

RDS MySQL Database in Private Subnets with Multi-AZ deployment

Security

Security Groups, NACLs, IAM Roles, Parameter Store for secrets

Project Structure

```
2-tier-architecture/
├── main.tf # Main configuration
├── variables.tf # Input variables
├── outputs.tf # Output values
├── terraform.tfvars # Variable values
├── modules/
│   ├── networking/ # VPC, subnets, gateways
│   ├── security/ # Security groups
│   ├── compute/ # EC2, ALB, ASG
│   └── database/ # RDS configuration
└── userdata/
    └── web_server.sh # EC2 initialization script
```

Key Features

- **High Availability:** Multi-AZ deployment across 2 availability zones
- **Scalability:** Auto Scaling Group for web tier
- **Security:** Database in private subnets, security groups
- **Load Balancing:** Application Load Balancer with health checks
- **Monitoring:** CloudWatch integration

Prepared By: Rashi Rana
Corporate Trainer

2-Tier Architecture Implementation

Database Configuration

RDS Subnet Group

```
resource "aws_db_subnet_group" "main" {
  name = "${var.environment}-db-subnet-group"
  subnet_ids = var.private_subnet_ids

  tags = {
    Name = "${var.environment}-db-subnet-group"
    Environment = var.environment
  }
}
```

RDS Instance

```
resource "aws_db_instance" "main" {
  identifier = "${var.environment}-database"

  engine = "mysql"
  engine_version = "8.0"
  instance_class = "db.t3.micro"

  allocated_storage = 20
  max_allocated_storage = 100
  storage_type = "gp2"
  storage_encrypted = true

  db_name = var.database_name
  username = var.database_username
  password = var.database_password

  vpc_security_group_ids =
```

```
[aws_security_group.database.id]
  db_subnet_group_name =
aws_db_subnet_group.main.name

  backup_retention_period = 7
  backup_window = "03:00-04:00"
  maintenance_window = "sun:04:00-sun:05:00"

  multi_az = true
  publicly_accessible = false
  skip_final_snapshot = true

  tags = {
    Name = "${var.environment}-database"
    Environment = var.environment
  }
}
```

Auto Scaling Group

```
resource "aws_launch_template" "web" {
  name_prefix = "${var.environment}-web-"
  image_id = data.aws_ami.amazon_linux.id
  instance_type = var.instance_type
  key_name = var.key_name

  vpc_security_group_ids =
[aws_security_group.web.id]

  user_data =
base64encode(templatefile("${path.module}/userdata/web_
{
    db_endpoint = aws_db_instance.main.endpoint
    db_name = var.database_name
  }))

  tag_specifications {
    resource_type = "instance"
    tags = {
      Name = "${var.environment}-web-server"
      Environment = var.environment
    }
  }
```

```
}

resource "aws_autoscaling_group" "web" {
  name = "${var.environment}-web-asg"
  vpc_zone_identifier = var.public_subnet_ids
  target_group_arns = [aws_lb_target_group.web.arn]
  health_check_type = "ELB"

  min_size = 2
  max_size = 6
  desired_capacity = 2

  launch_template {
    id = aws_launch_template.web.id
    version = "$Latest"
  }

  tag {
    key = "Name"
    value = "${var.environment}-web-asg"
    propagate_at_launch = false
  }
}
```

Prepared By: Rashi Rana
Corporate Trainer

Summary and Best Practices

Key Concepts Covered

- Infrastructure as Code principles and benefits
- Terraform fundamentals: providers, resources, variables, outputs
- Terraform workflow: init, plan, apply, destroy
- Advanced features: modules, state management, data sources
- Practical implementation of AWS infrastructure
- 2-tier architecture deployment with best practices

Terraform Best Practices

Code Organization

Use modules, consistent naming, proper file structure, and documentation

State Management

Remote backend, state locking, separate environments, regular backups

Security

No hardcoded secrets, least privilege access, encrypted

Version Control

Git workflows, .gitignore, lock files, semantic state, secure backends

versioning for modules

Production Considerations

- **Environment Separation:** Separate state files and configurations for dev/staging/prod
- **CI/CD Integration:** Automated planning and applying in pipelines
- **Monitoring:** CloudWatch, logging, and alerting for infrastructure changes
- **Disaster Recovery:** Multi-region deployments and backup strategies
- **Cost Optimization:** Resource tagging, right-sizing, and automated cleanup

Next Steps

1. **Advanced Terraform:** Workspaces, dynamic blocks, for_each loops
2. **Testing:** Terratest, kitchen-terraform, policy as code
3. **Multi-Cloud:** Azure, GCP provider usage
4. **Terraform Cloud:** Remote execution, policy enforcement
5. **Integration:** Ansible, Kubernetes, service mesh

> **Remember:** Infrastructure as Code is not just about tools—it's about culture, collaboration, and treating infrastructure with the same discipline as application code.

Prepared By: Rashi Rana
Corporate Trainer