2024

Terraform

Architecting IAC Solutions for AWS

Contents

Introduction to Infrastructure as Code (IaC):

Understanding the benefits of IaC

1. Consistency: Use declarative scripts to ensure identical infrastructure across environments.
   Example: Deploy identical environments for development, staging, and production.

2. Automation: Reduces manual configuration tasks.
   Example: Automate the provisioning of a new server with predefined specs.

3. Version Control: Infrastructure can be tracked and rolled back via Git.
   Example: Use GitHub to track updates to your infrastructure scripts.

4. Scalability: Easily replicate or scale infrastructure.
   Example: Scale a load balancer configuration to handle increased traffic.

5. Cost Management: Better visibility into provisioned resources.
   Example: Monitor and destroy unused resources to save costs.

Comparing IaC tools (Terraform, CloudFormation, Ansible, etc.)

Terraform

- Strength: Cloud-agnostic.
- Example: Use Terraform to provision EC2 instances on AWS and Compute Engine on GCP.

AWS CloudFormation

- Strength: AWS-specific.
- Example: Define an S3 bucket, an EC2 instance, and a VPC in YAML/JSON.

Ansible

- Strength: Configuration management.
- Example: Install software on an already-provisioned EC2 instance.

Pulumi

- Strength: Uses familiar programming languages.
- Example: Write Python code to create an AWS Lambda function.

Introduction to Terraform:

Core concepts: modules, providers, states, and workspaces

a) Modules: Reusable, self-contained infrastructure components.
   Example: A module for creating a VPC with subnets and routing tables.

b) Providers: Plugins that interact with cloud providers.
   Example: provider "aws" { region = "us-east-1" } to configure AWS as a provider.

c) State Files: Keep track of infrastructure and its current state.
   Example: A terraform.tfstate file showing deployed EC2 instances.

d) Workspaces: Isolate state files for different environments.
Example: Use default for staging and prod workspace for production.

Installation and configuration of Terraform

1. Download Terraform from the official site.
2. Add it to the system PATH.
3. Verify the installation:

```
terraform --version
```

Writing basic Terraform configuration files

Terraform uses a declarative language to define infrastructure as code. A Terraform configuration file typically consists of provider configuration, resource definitions, and optional variables, outputs, and modules.

Step 1: Set Up Terraform Environment
Install Terraform:

a) Download Terraform from the Terraform website.
b) Add Terraform to your system's PATH.

Create a Directory:

a) Create a folder to store your Terraform configuration files. For example:

```
mkdir terraform-basic-config
cd terraform-basic-config
```

Step 2: Create the main.tf File

a) Create a file named main.tf in your project directory. This file will contain the configuration.

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

# Resource Configuration: Create an EC2 Instance
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with a valid AMI ID
  instance_type = "t2.micro"

  tags = {
    Name = "BasicTerraformInstance"
  }
}
```

Step 3: Initialize Terraform

a) Run the following command to download the necessary provider plugins and prepare the directory:

```
terraform init
```

*This initializes Terraform and downloads the AWS provider plugin.

Step 4: Validate the Configuration

a) Check if your configuration is valid by running:

```
terraform validate
```

Output:
Success! The configuration is valid.

Step 5: Plan the Infrastructure

a) Preview the changes Terraform will make:

```
terraform plan
```

Output includes the resources Terraform will create (e.g., EC2 instance).

Step 6: Apply the Configuration

a) Deploy the defined infrastructure:

```
terraform apply
```

*Type yes when prompted.

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

Step 7: Verify the Deployed Resource

a) Log in to your AWS Management Console.
b) Navigate to the EC2 section in the specified region (us-east-1) to confirm the instance has been created.

Step 8: Clean Up

a) Destroy the resources when you're done:

```
terraform destroy
```

Type yes to confirm.

Additional Examples

Example 1: Create an S3 Bucket

```
resource "aws_s3_bucket" "example" {
  bucket = "my-unique-terraform-bucket"
  acl    = "private"
}
```

Example 2: Use Variables

```
variable "region" {
  default = "us-east-1"
}

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

Example 3: Add an EBS Volume

```
resource "aws_ebs_volume" "example" {
  availability_zone = "us-east-1a"
  size              = 10
}
```

Example 4: Use Output Values

```
output "instance_id" {
  value = aws_instance.example.id
}
```

Example 5: Use a Data Source

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

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

resource "aws_instance" "example" {
  ami           = data.aws_ami.latest.id
  instance_type = "t2.micro"
}
```

AWS Fundamentals:

Overview of AWS services

- *See attached AWS Fundamentals Trainer's Guide for this topic.*
- *Topic will be discussed by trainer separately*

Key concepts: regions, availability zones, EC2 instances, S3 buckets, VPCs, security groups, IAM roles and policies

- *See attached AWS Fundamentals Trainer's Guide for this topic.*
- *Topic will be discussed by trainer separately*

Hands-on Lab: Creating a Simple EC2 Instance

Set Up the Project Directory

Create a new folder for your Terraform project:

```
mkdir terraform-ec2-lab
cd terraform-ec2-lab
```

Inside this folder, create a new file named main.tf:

```
touch main.tf
```

Write the Terraform Configuration File

Edit the main.tf file and add the following content:

main.tf Content:

```
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1" # Replace with your desired AWS region
}

# Create a Key Pair
resource "aws_key_pair" "terraform_key" {
  key_name   = "terraform-key"
  public_key = file("~/.ssh/id_rsa.pub") # Use your local SSH public key file
}

# Create a Security Group
resource "aws_security_group" "allow_ssh" {
  name_prefix = "allow-ssh-"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

# Create an EC2 Instance
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with a valid AMI ID for your region
  instance_type = "t2.micro"

  key_name               = aws_key_pair.terraform_key.key_name
  security_groups        = [aws_security_group.allow_ssh.name]

  tags = {
    Name = "Terraform-EC2-Instance"
  }
}
```

Initialize the Terraform Working Directory

Run the following command to initialize Terraform and download the necessary provider plugins:

```
terraform init
```

Expected Output:
Terraform has been successfully initialized!

Validate the Configuration

Validate the configuration to ensure there are no syntax errors:

```
terraform validate
```

Expected Output:
Success! The configuration is valid.

Plan the Infrastructure Changes

Run a terraform plan to preview the infrastructure changes Terraform will make:

```
terraform plan
```

Expected Output:
A list of resources to be created, such as the EC2 instance, security group, and key pair.

Apply the Configuration

Deploy the infrastructure:

```
terraform apply
```

- ✓ Terraform will show you a plan similar to the one generated by terraform plan.
- ✓ Type yes to confirm and create the resources.

Expected Output:
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Verify the EC2 Instance

1) Log in to the AWS Management Console.
2) Navigate to EC2 > Instances in the us-east-1 region (or the region you specified).
3) Confirm that the EC2 instance is running, and check its public IP address.

Clean Up Resources

When you're done, destroy the infrastructure to avoid incurring costs:

```
terraform destroy
```

*Type yes to confirm resource deletion.

Verify the output:
Destroy complete! Resources: 3 destroyed.

Key Features Demonstrated:
Provider Configuration: Set up AWS as the provider.

Resource Management:
- Created a key pair for SSH access.
- Configured a security group to allow SSH traffic.
- Provisioned an EC2 instance using a specific AMI.

Lifecycle Commands:

terraform init	-to initialize the environment.
terraform plan	-to preview changes.
terraform apply	-to deploy the infrastructure.
terraform destroy	-to clean up resources.

Deep Dive into Terraform:

Variables and data sources

Variables
Variables allow dynamic values in your Terraform configuration.

Define a Variable:
In a file called variables.tf:

```
variable "instance_type" {
  default = "t2.micro"
  description = "The type of EC2 instance to create"
}
```

Use the Variable:
In your main.tf file:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type
}
```

Set the Variable Value:

```
terraform apply -var="instance_type=t3.medium"
```

Or use a terraform.tfvars file:

```
instance_type = "t3.medium"
```

Data Sources
Data sources fetch information about existing resources.

Define a Data Source:

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

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

Use the Data Source:

```
resource "aws_instance" "example" {
  ami           = data.aws_ami.latest.id
  instance_type = "t2.micro"
}
```

Example 1: Dynamic EC2 Instance Type
This example dynamically sets the instance type using variables and fetches the latest Amazon Linux AMI using a data source.

Configuration:

```
# variables.tf
variable "instance_type" {
  default     = "t2.micro"
  description = "The type of EC2 instance"
}
```

```
# data.tf
data "aws_ami" "latest" {
  most_recent = true
  owners      = ["amazon"]

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

```
# main.tf
resource "aws_instance" "example" {
  ami           = data.aws_ami.latest.id
  instance_type = var.instance_type

  tags = {
    Name = "DynamicEC2Instance"
```

```
  }
}
```

Steps:

- Change the instance_type dynamically via CLI or terraform.tfvars.
- The AMI is fetched dynamically using the data source.

Example 2: Dynamic S3 Bucket Name
This example creates an S3 bucket with a name derived from a variable.

Configuration:

```
# variables.tf
variable "project_name" {
  description = "Project name for the bucket"
  default     = "my-project"
}
```

```
# main.tf
resource "aws_s3_bucket" "example" {
  bucket = "${var.project_name}-bucket"
  acl    = "private"
}

data "aws_region" "current" {}

output "bucket_region" {
  value = data.aws_region.current.name
}
```

Steps:

- Change the project_name variable to modify the bucket name.
- Use the aws_region data source to fetch the current region for display.

Modules and their organization

Modules help organize reusable Terraform code.

Create a Module

Create a directory for your module:

```
mkdir -p modules/ec2
cd modules/ec2
```

Add main.tf:

```
resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type

  tags = {
    Name = var.instance_name
```

```
  }
}
```

Add variables.tf:

```
variable "ami" {}
variable "instance_type" {
  default = "t2.micro"
}
variable "instance_name" {}
```

Add outputs.tf:

```
output "instance_id" {
  value = aws_instance.example.id
}
```

<u>Use the Module</u>
In your main project directory, reference the module:

```
module "ec2_instance" {
  source        = "./modules/ec2"
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  instance_name = "MyEC2Instance"
}
```

Run Terraform:

```
terraform init
terraform apply
```

Example 1: VPC Module
This example uses a module to create a VPC with subnets and a route table.

Module (in modules/vpc/main.tf):

```
resource "aws_vpc" "example" {
  cidr_block = var.cidr_block
}

resource "aws_subnet" "example" {
  count      = var.subnet_count
  vpc_id     = aws_vpc.example.id
  cidr_block = cidrsubnet(var.cidr_block, 8, count.index)
}

variable "cidr_block" {}
variable "subnet_count" {}
```

Main Configuration:

```
module "vpc" {
  source     = "./modules/vpc"
  cidr_block  = "10.0.0.0/16"
  subnet_count = 2
}
```

Example 2: EC2 Module
Reuse an EC2 module for different environments.

Module (in modules/ec2/main.tf):

```
resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.instance_type

  tags = {
    Name = var.instance_name
  }
}

variable "ami" {}
variable "instance_type" {}
variable "instance_name" {}
```

Main Configuration:

```
module "staging_ec2" {
  source        = "./modules/ec2"
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  instance_name = "StagingInstance"
}

module "prod_ec2" {
  source        = "./modules/ec2"
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.medium"
  instance_name = "ProdInstance"
}
```

State management and remote backends

Understand State

Terraform tracks infrastructure in a .tfstate file. By default, this file is stored locally.

Enable a Remote Backend

Configure backend in your main.tf:

```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "state/terraform.tfstate"
```

```
    region        = "us-east-1"
  }
}
```

Initialize Terraform:

```
terraform init
```

*This migrates the local state to the remote backend.

Example 1: Remote Backend in S3

This example sets up Terraform state storage in an S3 bucket.

Main Configuration:

```
terraform {
  backend "s3" {
    bucket        = "my-terraform-state"
    key           = "state/terraform.tfstate"
    region        = "us-east-1"
  }
}

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

Steps:

- Create the S3 bucket my-terraform-state.
- Run terraform init to migrate the local state to the S3 backend.

Example 2: Shared State for Multiple Teams

Use a remote backend and locking for collaboration.

Main Configuration:

```
terraform {
  backend "s3" {
    bucket        = "shared-terraform-state"
    key           = "dev/terraform.tfstate"
    region        = "us-east-1"
    dynamodb_table = "terraform-locking"
  }
}
```

Steps:

- Configure DynamoDB table terraform-locking for state locking.
- Teams working on the same configuration will avoid race conditions.

Terraform providers and their usage

What Are Providers?

Providers are plugins used to interact with cloud platforms and other APIs.

Configure a Provider

Add a provider block:

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

Specify provider requirements in versions.tf:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}
```

Use Multiple Providers

Define multiple providers:

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

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

Reference a provider in resources:

```
resource "aws_instance" "example" {
  provider      = aws.primary
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Example 1: AWS and GCP Providers

Provision an EC2 instance in AWS and a VM in Google Cloud.

Configuration:

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

provider "google" {
  project = "my-gcp-project"
```

```
  region  = "us-central1"
}

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

resource "google_compute_instance" "example" {
  name         = "gcp-vm"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }

  network_interface {
    network = "default"
  }
}
```

Example 2: Multiple AWS Regions
Deploy instances in different AWS regions using provider aliases.

Configuration:

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

provider "aws" {
  alias  = "us_west"
  region = "us-west-2"
}

resource "aws_instance" "east_instance" {
  provider      = aws.us_east
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

resource "aws_instance" "west_instance" {
  provider      = aws.us_west
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

AWS Networking with Terraform:

Creating VPCs, subnets, and internet gateways

A VPC (Virtual Private Cloud) is a logically isolated network for AWS resources. Subnets divide the VPC into smaller segments, and Internet Gateways provide internet connectivity.

Example 1: Creating a Simple VPC with Subnets

Configuration:

```
resource "aws_vpc" "example" {
  cidr_block         = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "MyVPC"
  }
}

resource "aws_subnet" "example" {
  vpc_id          = aws_vpc.example.id
  cidr_block       = "10.0.1.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "MySubnet"
  }
}
```

Steps:

- Run terraform apply.
- Check the VPC and subnet in the AWS console.

Example 2: VPC with Two Subnets

Configuration:

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "MyVPC"
  }
}

resource "aws_subnet" "public_subnet" {
  vpc_id          = aws_vpc.example.id
  cidr_block       = "10.0.1.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PublicSubnet"
  }
}

resource "aws_subnet" "private_subnet" {
```

```
  vpc_id          = aws_vpc.example.id
  cidr_block      = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PrivateSubnet"
  }
}
```

Example 3: Adding an Internet Gateway
Configuration:

```
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "MyVPC"
  }
}

resource "aws_internet_gateway" "example" {
  vpc_id = aws_vpc.example.id
  tags = {
    Name = "MyInternetGateway"
  }
}

resource "aws_subnet" "example" {
  vpc_id          = aws_vpc.example.id
  cidr_block      = "10.0.1.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "MySubnet"
  }
}
```

Configuring route tables and network ACLs

Example 1: Adding a Route Table for Internet Access
Configuration:

```
resource "aws_route_table" "example" {
  vpc_id = aws_vpc.example.id

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

  tags = {
    Name = "MyRouteTable"
  }
}
```

```
resource "aws_route_table_association" "example" {
  subnet_id      = aws_subnet.example.id
  route_table_id = aws_route_table.example.id
}
```

Example 2: Creating a Network ACL

Configuration:

```
resource "aws_network_acl" "example" {
  vpc_id = aws_vpc.example.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    rule_no     = 100
    cidr_block  = "0.0.0.0/0"
    action      = "allow"
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    rule_no     = 200
    cidr_block  = "0.0.0.0/0"
    action      = "allow"
  }

  tags = {
    Name = "MyNetworkACL"
  }
}
```

Example 3: Associating a Subnet with a Network ACL

Configuration:

```
resource "aws_network_acl_association" "example" {
  subnet_id      = aws_subnet.example.id
  network_acl_id = aws_network_acl.example.id
}
```

Assigning public and private IP addresses to EC2 instances

Example 1: Assigning a Public IP Address

Configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.example.id
  associate_public_ip_address = true
```

```
  tags = {
    Name = "PublicInstance"
  }
}
```

Example 2: Assigning a Private IP Address
Configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.private_subnet.id
  private_ip    = "10.0.2.10"

  tags = {
    Name = "PrivateInstance"
  }
}
```

Example 3: Creating an EC2 Instance in Both Public and Private Subnets
Configuration:

```
module "public_instance" {
  source              = "./modules/ec2"
  subnet_id           = aws_subnet.public_subnet.id
  associate_public_ip_address = true
  instance_name       = "PublicInstance"
}

module "private_instance" {
  source              = "./modules/ec2"
  subnet_id           = aws_subnet.private_subnet.id
  private_ip          = "10.0.2.10"
  instance_name       = "PrivateInstance"
}
```

Hands-on Lab: Creating a VPC and Subnets

Set Up Your Project Directory
Create a new directory for your Terraform project:

```
mkdir terraform-vpc
cd terraform-vpc
```

Create the main Terraform configuration file:

```
touch main.tf
```

Write the Terraform Configuration File
Here's how to create a VPC with one public and one private subnet.

Complete Configuration (main.tf):

```
# Provider Configuration
provider "aws" {
  region = "us-east-1" # Specify your AWS region
}

# Create a VPC
resource "aws_vpc" "my_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "MyVPC"
  }
}

# Create an Internet Gateway
resource "aws_internet_gateway" "my_igw" {
  vpc_id = aws_vpc.my_vpc.id
  tags = {
    Name = "MyInternetGateway"
  }
}

# Create a Public Subnet
resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.my_vpc.id
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "PublicSubnet"
  }
}

# Create a Private Subnet
resource "aws_subnet" "private_subnet" {
  vpc_id            = aws_vpc.my_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PrivateSubnet"
  }
}

# Create a Route Table for Public Subnet
resource "aws_route_table" "public_route_table" {
```

```
  vpc_id = aws_vpc.my_vpc.id
  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.my_igw.id
  }
  tags = {
    Name = "PublicRouteTable"
  }
}

# Associate Public Subnet with the Route Table
resource "aws_route_table_association" "public_route_assoc" {
  subnet_id      = aws_subnet.public_subnet.id
  route_table_id = aws_route_table.public_route_table.id
}
```

Initialize Terraform

Run the following command to initialize Terraform. This downloads necessary provider plugins:

```
terraform init
```

Expected Output:
Terraform has been successfully initialized!

Plan the Infrastructure

Run terraform plan to preview the resources Terraform will create:

```
terraform plan
```

Expected Output:
A summary of the resources to be created, including the VPC, subnets, and internet gateway.

Apply the Configuration

Run the following command to create the VPC and subnets:

```
terraform apply
```

*When prompted, type yes to confirm.

Expected Output:
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Verify the Resources in AWS

a) Log in to the AWS Management Console.
b) Navigate to the VPC section.
c) Verify that:
   - ✓ A VPC with CIDR block 10.0.0.0/16 exists.
   - ✓ One public subnet (10.0.1.0/24) is associated with the internet gateway.
   - ✓ One private subnet (10.0.2.0/24) is created.

Clean Up Resources
To avoid unnecessary charges, destroy the resources when you're done:

```
terraform destroy
```

*Type yes to confirm.

Terraform Best Practices:

Modularizing Terraform configurations

Modularizing Terraform code involves breaking down the configuration into reusable, organized units called modules.

Create a Module
Modules are self-contained configurations.

Directory Structure:

```
terraform-project/
├── main.tf        # Main file to call modules
├── variables.tf   # Variables for the root module
├── outputs.tf     # Outputs for the root module
└── modules/
    └── vpc/
        ├── main.tf
        ├── variables.tf
        ├── outputs.tf
```

Module (modules/vpc):

```
# modules/vpc/main.tf:
resource "aws_vpc" "example" {
  cidr_block = var.cidr_block
  tags = {
    Name = var.name
  }
}

resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.example.id
  cidr_block        = var.public_subnet_cidr
  map_public_ip_on_launch = true
}
```

```
# modules/vpc/variables.tf:
variable "cidr_block" {}
variable "name" {}
variable "public_subnet_cidr" {}
```

```
# modules/vpc/outputs.tf:
output "vpc_id" {
  value = aws_vpc.example.id
}
```

```
output "public_subnet_id" {
  value = aws_subnet.public_subnet.id
}
```

Root Module (main.tf):

```
module "vpc" {
  source           = "./modules/vpc"
  cidr_block       = "10.0.0.0/16"
  name             = "MyVPC"
  public_subnet_cidr = "10.0.1.0/24"
}
```

Apply Modular Configuration

Initialize Terraform:

```
terraform init
```

Plan and apply:

```
terraform plan
terraform apply
```

Benefits of Modularization:

- ✓ Reusability: Modules can be reused across projects.
- ✓ Maintainability: Easier to manage smaller pieces of code.
- ✓ Team Collaboration: Different teams can work on separate modules.

Testing and validation of Terraform code

Testing and validating Terraform ensures infrastructure is deployed as expected and reduces risks.

Syntax Validation

Validate configuration files for syntax errors.

```
terraform validate
```

Pre-Deployment Testing

Plan Command: Preview the infrastructure changes.

```
terraform plan
```

Unit Testing with terratest

Use the terratest library for automated testing.

1) Install terratest in Go.
2) Write a test:

go

```
package test

import (
  "testing"
  "github.com/gruntwork-io/terratest/modules/terraform"
)
```

```
func TestTerraformVPC(t *testing.T) {
  options := &terraform.Options{
    TerraformDir: "../terraform-project",
  }

  defer terraform.Destroy(t, options)

  terraform.InitAndApply(t, options)

  vpcID := terraform.Output(t, options, "vpc_id")
  if vpcID == "" {
    t.Fatal("VPC ID is empty")
  }
}
```

Run the test:

```
go test -v
```

Linting with tflint

Install tflint and run it to check for potential errors.

```
tflint
```

Security considerations in Terraform

Secure State Files

Store State Remotely: Use secure backends like S3 with encryption:

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

*Encrypt State Locally: Use secure storage for local .tfstate files (e.g., disk encryption).

Manage Secrets

Avoid Hardcoding Secrets: Use environment variables or secret management tools like AWS Secrets Manager or HashiCorp Vault.

Example with Vault:

```
export VAULT_ADDR='http://127.0.0.1:8200'
export VAULT_TOKEN='my-token'
terraform plan
```

Least Privilege IAM Roles
Assign minimum permissions to the Terraform IAM user or role.

Use Security Scanners
Scan Terraform configurations with tools like:

```
checkov
tfsec
```

Version control for Terraform code

Use Git for Version Control
Initialize a Git repository:

```
git init
```

Track Terraform files:

```
git add main.tf variables.tf outputs.tf
git commit -m "Initial Terraform configuration"
```

Use Branching for Changes
Create a feature branch:

```
git checkout -b feature/add-vpc
```

Commit and push changes:

```
git commit -am "Add VPC configuration"
git push origin feature/add-vpc
```

Use .gitignore
Ignore sensitive files and local state files:

```
*.tfstate
*.tfstate.backup
.terraform/
```

Tagging for Releases
Tag stable versions of infrastructure:

```
git tag -a v1.0 -m "Stable release"
git push origin v1.0
```

Automate with CI/CD
Use GitHub Actions or Jenkins to automate Terraform workflows.
Example GitHub Action:

```
name: Terraform

on:
  push:
    branches:
      - main

jobs:
```

```
  terraform:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
      - name: Terraform Init
        run: terraform init
      - name: Terraform Plan
        run: terraform plan
```

Advanced AWS Services with Terraform:

Deploying S3 buckets and object storage

Step 1: Create an S3 Bucket

Add the following configuration to create an S3 bucket in main.tf:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-terraform-bucket-12345" # Replace with a unique bucket name
  acl    = "private"

  tags = {
    Name        = "MyS3Bucket"
    Environment = "Dev"
  }
}
```

Initialize Terraform:

```
terraform init
```

Apply the configuration:

```
terraform apply
```

Step 2: Enable Versioning and Logging

Update the configuration to enable versioning and logging:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-terraform-bucket-12345"
  acl    = "private"

  versioning {
    enabled = true
  }

  logging {
    target_bucket = "my-log-bucket"
    target_prefix = "logs/"
  }
```

```
  tags = {
    Name        = "MyS3Bucket"
    Environment = "Dev"
  }
}
```

Apply the updated configuration:

```
terraform apply
```

Step 3: Add an S3 Object

Add an object to the bucket:

```
resource "aws_s3_object" "example" {
  bucket = aws_s3_bucket.example.id
  key    = "example.txt"
  content = "Hello, Terraform!"
}
```

Apply the changes:

```
terraform apply
```

Creating IAM roles and policies

Step 1: Create an IAM Role

Add the following configuration:

```
resource "aws_iam_role" "example" {
  name = "example-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })
}
```

Apply the configuration:

```
terraform apply
```

Step 2: Create an IAM Policy

Add the following configuration:

```
resource "aws_iam_policy" "example_policy" {
  name        = "example-policy"
  description = "A policy to allow S3 access"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action   = "s3:*"
      Effect   = "Allow"
      Resource = "*"
    }]
  })
}
```

Apply the configuration:

```
terraform apply
```

Step 3: Attach the Policy to the Role

Add the following:

```
resource "aws_iam_role_policy_attachment" "example" {
  role       = aws_iam_role.example.name
  policy_arn = aws_iam_policy.example_policy.arn
}
```

Apply the configuration:

```
terraform apply
```

Configuring security groups and network ACLs

Step 1: Create a Security Group

Add the following configuration:

```
resource "aws_security_group" "example" {
  name        = "example-sg"
  description = "Allow SSH and HTTP"
  vpc_id      = aws_vpc.example.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
```

```
  }

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

  tags = {
    Name = "ExampleSecurityGroup"
  }
}
```

Apply the configuration:

```
terraform apply
```

Step 2: Create a Network ACL

Add the following configuration:

```
resource "aws_network_acl" "example" {
  vpc_id = aws_vpc.example.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    rule_no     = 100
    cidr_block  = "0.0.0.0/0"
    action      = "allow"
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    rule_no     = 200
    cidr_block  = "0.0.0.0/0"
    action      = "allow"
  }

  tags = {
    Name = "ExampleNetworkACL"
  }
}
```

Apply the configuration:

```
terraform apply
```

Automating deployments with Terraform Cloud

Step 1: Sign Up for Terraform Cloud

a) Create an account at Terraform Cloud.
b) Create a new workspace.

Step 2: Configure Terraform Cloud Backend

Add the following backend configuration in main.tf:

```
terraform {
  backend "remote" {
    organization = "your-organization-name"

    workspaces {
      name = "example-workspace"
    }
  }
}
```

Initialize Terraform:

```
terraform init
```

Step 3: Push Code to Version Control

Commit your Terraform configuration:

```
git init
git add .
git commit -m "Initial commit"
git push origin main
```

*Connect the Terraform Cloud workspace to your Git repository.

Step 4: Automate Plan and Apply

- Configure Terraform Cloud to automatically run plan and apply when changes are pushed to the repository.
- Monitor the deployment in the Terraform Cloud UI.

Hands-on Lab: Creating an S3 Bucket with Versioning

Write a Terraform Configuration File to Create an S3 Bucket with Versioning

Step 1: Set Up the Project Directory

Create a new directory for your Terraform project:

```
mkdir terraform-s3-bucket
cd terraform-s3-bucket
```

Create the main Terraform configuration file:

```
touch main.tf
```

Step 2: Add the Configuration in main.tf

Here's the Terraform configuration to create an S3 bucket with versioning:

```
# Configure the AWS Provider
provider "aws" {
  region = "us-east-1" # Replace with your preferred region
}

# Create the S3 Bucket
resource "aws_s3_bucket" "example" {
  bucket = "my-versioned-bucket-12345" # Replace with a unique bucket name
  acl    = "private"                # Set bucket ACL to private

  tags = {
    Name        = "VersionedS3Bucket"
    Environment = "Dev"
  }
}

# Enable Versioning on the Bucket
resource "aws_s3_bucket_versioning" "example" {
  bucket = aws_s3_bucket.example.id

  versioning_configuration {
    status = "Enabled" # Enable versioning
  }
}

# Add Output to Display the Bucket Name
output "bucket_name" {
  value = aws_s3_bucket.example.bucket
}
```

Apply the Configuration to Create the S3 Bucket

Step 1: Initialize Terraform

Run the following command to initialize the Terraform project and download the necessary provider plugins:

```
terraform init
```

Expected Output:
Terraform has been successfully initialized!

Step 2: Validate the Configuration

Check if the configuration is valid:

```
terraform validate
```

Expected Output:
Success! The configuration is valid.

Step 3: Plan the Infrastructure Changes
Preview the changes Terraform will make:

```
terraform plan
```

Expected Output:
A detailed summary of resources to be created, including the S3 bucket and its versioning configuration.

Step 4: Apply the Configuration
Run the following command to create the S3 bucket:

```
terraform apply
```

*When prompted, type yes to confirm.

Expected Output:
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Verify the S3 Bucket in AWS

a) Log in to the AWS Management Console.
b) Navigate to the S3 service.
c) Confirm that:
   - ✓ The bucket (my-versioned-bucket-12345) has been created.
   - ✓ Versioning is enabled under the Properties tab for the bucket.

Clean Up Resources (Optional)
To avoid unnecessary costs, destroy the resources when you're done:

```
terraform destroy
```

Expected Output:

```
Destroy complete! Resources: 2 destroyed.
```

Terraform and AWS Security:

Implementing security best practices in Terraform configurations

<u>Best Practice 1: Avoid Hardcoding Secrets</u>
Problem: Storing sensitive data like API keys or passwords directly in configuration files.
Solution: Use environment variables or secret management tools.

Example Using Environment Variables:

```
provider "aws" {
  region     = "us-east-1"
  access_key = var.aws_access_key
  secret_key = var.aws_secret_key
}

variable "aws_access_key" {}
variable "aws_secret_key" {}
```

Run Terraform with environment variables:

```
export TF_VAR_aws_access_key="your-access-key"
export TF_VAR_aws_secret_key="your-secret-key"
terraform apply
```

<u>Best Practice 2: Use Remote Backends to Store State</u>
Problem: Local .tfstate files can be accessed or modified accidentally.
Solution: Use remote backends like S3 with encryption.

Example Using S3 as a Backend:

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

Initialize Terraform with the backend:

```
terraform init
```

<u>Best Practice 3: Scan Configurations for Security Issues</u>
Use tools like tflint, tfsec, or checkov to identify vulnerabilities.

```
tflint
tfsec .
```

<u>Best Practice 4: Use Least Privilege Principle for IAM Roles</u>
Grant minimal permissions required for Terraform and AWS resources (discussed in Section 2).

Securing AWS resources with IAM roles and policies

Step 1: Create an IAM Role
Configuration:

```
resource "aws_iam_role" "example" {
  name = "example-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
      Action = "sts:AssumeRole"
    }]
  })
}
```

Step 2: Create a Policy for S3 Access

Configuration:

```
resource "aws_iam_policy" "example_policy" {
  name        = "example-policy"
  description = "Allow S3 access"
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Effect   = "Allow"
      Action   = ["s3:ListBucket", "s3:GetObject", "s3:PutObject"]
      Resource = ["arn:aws:s3:::example-bucket", "arn:aws:s3:::example-bucket/*"]
    }]
  })
}
```

Step 3: Attach the Policy to the Role

Configuration:

```
resource "aws_iam_role_policy_attachment" "example_attachment" {
  role       = aws_iam_role.example.name
  policy_arn = aws_iam_policy.example_policy.arn
}
```

Step 4: Verify and Apply the Configuration

Initialize Terraform:

```
terraform init
```

Plan and apply:

```
terraform plan
terraform apply
```

Using security groups to control network traffic

Security groups act as virtual firewalls to control inbound and outbound traffic to AWS resources.

Step 1: Create a Security Group

Configuration:

```
resource "aws_security_group" "example" {
  name        = "example-sg"
  description = "Allow SSH and HTTP"
  vpc_id      = aws_vpc.example.id

  # Allow inbound SSH traffic
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
  # Allow inbound HTTP traffic
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Allow all outbound traffic
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "ExampleSecurityGroup"
  }
}
```

Step 2: Associate Security Group with EC2 Instance

Configuration:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  security_groups = [
    aws_security_group.example.name
  ]

  tags = {
    Name = "ExampleInstance"
  }
}
```

Step 3: Verify and Apply the Configuration

Initialize Terraform:

```
terraform init
```

Plan and apply:

```
terraform plan
terraform apply
```

Real-world Use Cases:

Deploying multi-tier applications on AWS

Objective: Deploy a simple multi-tier application with a web tier (EC2) and a database tier (RDS).

Step 1: Define the Project Structure

Directory Layout:

```
multi-tier-app/
├── main.tf        # Main configuration
├── variables.tf    # Variables for reusability
├── outputs.tf      # Outputs to access resources
```

Step 2: Create the VPC and Subnets

Add the following configuration in main.tf:

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

resource "aws_vpc" "app_vpc" {
  cidr_block          = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "AppVPC"
  }
}

resource "aws_subnet" "public_subnet" {
  vpc_id           = aws_vpc.app_vpc.id
  cidr_block       = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "PublicSubnet"
  }
}

resource "aws_subnet" "private_subnet" {
  vpc_id           = aws_vpc.app_vpc.id
  cidr_block       = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PrivateSubnet"
  }
}
```

Step 3: Launch the Web Tier

a) Add an EC2 instance in the public subnet for the web server.
b) Create a security group to allow HTTP and SSH traffic.

```
resource "aws_security_group" "web_sg" {
  vpc_id = aws_vpc.app_vpc.id
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnet.id
  security_groups = [aws_security_group.web_sg.name]
  tags = {
    Name = "WebServer"
  }
}
```

Step 4: Launch the Database Tier

Use RDS in the private subnet for the database tier.

```
resource "aws_db_instance" "db_instance" {
  allocated_storage    = 20
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t2.micro"
  name                 = "appdb"
  username             = "admin"
  password             = "password123"
  vpc_security_group_ids = [aws_security_group.web_sg.id]
  db_subnet_group_name = aws_db_subnet_group.db_subnet_group.name
}

resource "aws_db_subnet_group" "db_subnet_group" {
```

```
  name      = "db-subnet-group"
  subnet_ids = [aws_subnet.private_subnet.id]
  tags = {
    Name = "DBSubnetGroup"
  }
}
```

Step 5: Apply the Configuration

Initialize Terraform:

```
terraform init
```

Plan the infrastructure:

```
terraform plan
```

Apply the configuration:

```
terraform apply
```

Automating infrastructure for CI/CD pipelines

Objective: Create a CI/CD pipeline with CodePipeline, CodeBuild, and S3.

Step 1: Create an S3 Bucket for CodePipeline

```
resource "aws_s3_bucket" "codepipeline_bucket" {
  bucket = "my-codepipeline-bucket"
  acl    = "private"
  tags = {
    Name = "CodePipelineBucket"
  }
}
```

Step 2: Define the CodePipeline

```
resource "aws_codepipeline" "my_pipeline" {
  name     = "my-pipeline"
  role_arn = aws_iam_role.codepipeline_role.arn

  artifact_store {
    type     = "S3"
    location = aws_s3_bucket.codepipeline_bucket.bucket
  }

  stage {
    name = "Source"

    action {
      name             = "SourceAction"
      category         = "Source"
      owner            = "AWS"
      provider         = "S3"
      version          = "1"
      output_artifacts = ["SourceOutput"]
```

```
    configuration = {
      S3Bucket = aws_s3_bucket.codepipeline_bucket.bucket
      S3ObjectKey = "source.zip"
    }
   }
  }

  stage {
   name = "Build"

   action {
    name            = "BuildAction"
    category        = "Build"
    owner           = "AWS"
    provider        = "CodeBuild"
    input_artifacts  = ["SourceOutput"]
    output_artifacts = ["BuildOutput"]

    configuration = {
      ProjectName = aws_codebuild_project.my_project.name
    }
   }
  }
}
```

Step 3: Define the CodeBuild Project

```
resource "aws_codebuild_project" "my_project" {
  name          = "MyProject"
  description   = "My CodeBuild Project"
  build_timeout = 5
  service_role  = aws_iam_role.codebuild_role.arn

  source {
   type = "S3"
   location = "${aws_s3_bucket.codepipeline_bucket.bucket}/source.zip"
  }

  artifacts {
   type = "NO_ARTIFACTS"
  }

  environment {
   compute_type                = "BUILD_GENERAL1_SMALL"
   image                       = "aws/codebuild/standard:5.0"
   type                        = "LINUX_CONTAINER"
   privileged_mode             = true
  }
}
```

Implementing infrastructure as code for disaster recovery

Objective: Implement automated recovery for critical infrastructure using Terraform.

Step 1: Define Remote State

Use S3 for state management to track the primary infrastructure.

```
terraform {
  backend "s3" {
    bucket       = "primary-state"
    key          = "terraform.tfstate"
    region       = "us-east-1"
  }
}
```

Step 2: Define Secondary Resources

Use data sources to fetch the current infrastructure.
Create duplicates in a disaster recovery region.

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

provider "aws" {
  alias  = "dr"
  region = "us-west-2"
}

data "aws_instance" "primary_instance" {
  provider = aws.primary
  instance_id = "i-1234567890abcdef"
}

resource "aws_instance" "dr_instance" {
  provider      = aws.dr
  ami           = data.aws_instance.primary_instance.ami
  instance_type = data.aws_instance.primary_instance.instance_type
  tags = {
    Name = "DRInstance"
  }
}
```

Step 3: Apply Disaster Recovery Plan

Initialize Terraform with the new region:

```
terraform init
```

Apply the disaster recovery configuration:

```
terraform apply
```

Hands-on Lab: Creating a Multi-Tier Application

Objective
Deploy a multi-tier application on AWS consisting of:

- Web Tier: EC2 instances behind a Load Balancer.
- Database Tier: RDS MySQL instance.

Step 1: Define the Project Structure
Directory Layout:

```
multi-tier-app/
├── main.tf        # Main configuration file
├── variables.tf   # Variables for reusability
├── outputs.tf     # Outputs to access resources
```

Step 2: Write the Terraform Configuration File

<u>Configure the Provider</u>
Add the AWS provider in main.tf:

```
provider "aws" {
  region = "us-east-1" # Specify your preferred region
}
```

<u>Create a VPC and Subnets</u>
Define a VPC with public and private subnets.

```
resource "aws_vpc" "app_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true
  tags = {
    Name = "AppVPC"
  }
}

resource "aws_subnet" "public_subnet" {
  vpc_id            = aws_vpc.app_vpc.id
  cidr_block        = "10.0.1.0/24"
  map_public_ip_on_launch = true
  availability_zone = "us-east-1a"
  tags = {
    Name = "PublicSubnet"
  }
}

resource "aws_subnet" "private_subnet" {
  vpc_id            = aws_vpc.app_vpc.id
  cidr_block        = "10.0.2.0/24"
  availability_zone = "us-east-1a"
  tags = {
    Name = "PrivateSubnet"
```

```
  }
}
```

Configure the Security Groups

Define security groups for the load balancer, EC2 instances, and RDS.

```
resource "aws_security_group" "web_sg" {
  vpc_id = aws_vpc.app_vpc.id
  description = "Allow HTTP and SSH traffic"

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

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

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

  tags = {
    Name = "WebSecurityGroup"
  }
}

resource "aws_security_group" "db_sg" {
  vpc_id = aws_vpc.app_vpc.id
  description = "Allow MySQL traffic"

  ingress {
    from_port   = 3306
    to_port     = 3306
    protocol    = "tcp"
    security_groups = [aws_security_group.web_sg.id]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
```

```
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "DBSecurityGroup"
  }
}
```

<u>Deploy the EC2 Instances</u>
Launch EC2 instances in the public subnet.

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0" # Replace with a valid AMI ID
  instance_type = "t2.micro"
  subnet_id     = aws_subnet.public_subnet.id
  security_groups = [aws_security_group.web_sg.name]

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

<u>Deploy a Load Balancer</u>
Create an Application Load Balancer for the EC2 instances.

```
resource "aws_lb" "app_lb" {
  name               = "app-load-balancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.web_sg.id]
  subnets            = [aws_subnet.public_subnet.id]

  tags = {
    Name = "AppLoadBalancer"
  }
}

resource "aws_lb_target_group" "web_tg" {
  name     = "web-target-group"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.app_vpc.id
}

resource "aws_lb_listener" "http_listener" {
  load_balancer_arn = aws_lb.app_lb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.web_tg.arn
```

```
  }
}

resource "aws_lb_target_group_attachment" "web_target" {
  target_group_arn = aws_lb_target_group.web_tg.arn
  target_id        = aws_instance.web_server.id
  port             = 80
}
```

Deploy an RDS Database

Create an RDS MySQL instance in the private subnet.

```
resource "aws_db_instance" "app_db" {
  allocated_storage    = 20
  engine               = "mysql"
  engine_version       = "8.0"
  instance_class       = "db.t2.micro"
  name                 = "appdb"
  username             = "admin"
  password             = "password123"
  vpc_security_group_ids = [aws_security_group.db_sg.id]
  db_subnet_group_name = aws_db_subnet_group.db_subnet_group.name
}

resource "aws_db_subnet_group" "db_subnet_group" {
  name       = "db-subnet-group"
  subnet_ids = [aws_subnet.private_subnet.id]
  tags = {
    Name = "DBSubnetGroup"
  }
}
```

Add Outputs

Define outputs to display key information.

```
output "load_balancer_dns" {
  value = aws_lb.app_lb.dns_name
}

output "database_endpoint" {
  value = aws_db_instance.app_db.endpoint
}
```

Step 3: Apply the Configuration to Deploy the Application

Initialize Terraform

Run the following command to initialize Terraform and download required providers:

```
terraform init
```

Validate the Configuration

Validate the syntax and configuration:

```
terraform validate
```

Plan the Deployment

Generate a detailed plan to verify resources:

```
terraform plan
```

Apply the Configuration

Deploy the infrastructure:

```
terraform apply
```

*When prompted, type yes to confirm.

Step 4: Verify the Deployment

Access the Load Balancer DNS:

a) Copy the DNS name from the output of terraform apply or from the AWS console.
b) Open it in a browser to verify the web server is accessible.

Verify the RDS database:

a) Use the endpoint from the Terraform output.
b) Connect using a MySQL client with the username and password defined in the configuration.