Comprehensive Terraform Interview Questions - Categorized by Importance (Interview-Ready)

Category 1: Core Terraform Concepts (HIGHEST PRIORITY - Must Know)

1.1 MOST CRITICAL (Must Know for Any Terraform Role)

Q1: What is Terraform and why is it important for Infrastructure as Code?

How to Answer in Interview:
"Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp that allows you to define, provision, and manage infrastructure using declarative configuration files. Key benefits include:

- *Consistency: Infrastructure is defined as code, ensuring reproducible deployments*
- *Version control: Infrastructure changes can be tracked and reviewed*
- *Multi-cloud support: Works across AWS, Azure, GCP, and 100+ providers*
- *Automation: Reduces manual errors and speeds up deployments*
- *Planning: Shows what changes will be made before applying them*

For example, instead of manually clicking through AWS console to create resources, I write Terraform code that consistently creates the same infrastructure every time."

Q2: Explain the Terraform workflow and core commands.

How to Answer in Interview:
"The Terraform workflow follows a simple three-step process:

1. *Write: Define infrastructure in .tf files using HCL (HashiCorp Configuration Language)*
2. *Plan: Use terraform plan to preview changes before applying*
3. *Apply: Use terraform apply to create/modify infrastructure*

Core commands I use daily:

- *terraform init: Initialize working directory and download providers*
- *terraform plan: Show execution plan without making changes*
- *terraform apply: Execute the plan and create/update resources*
- *terraform destroy: Remove all managed infrastructure*
- *terraform validate: Check configuration syntax*

This workflow ensures I never make surprise changes to production infrastructure."

Code Example:

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

provider "aws" {
  region = var.aws_region
}

# Variable definition
variable "aws_region" {
  description = "AWS region for resources"
  type        = string
  default     = "us-west-2"
}

# Resource definition
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1d0"
  instance_type = "t3.micro"

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

# Output definition
output "instance_ip" {
  description = "Public IP of the web server"
  value       = aws_instance.web_server.public_ip
}
```

Practice Questions:

1. Write a basic Terraform configuration to create an S3 bucket.
2. Explain the difference between terraform plan and terraform apply.
3. What happens if you run terraform apply twice with the same configuration?

Q3: What is Terraform state and why is it crucial?

How to Answer in Interview:
"Terraform state is a JSON file that tracks the current state of your infrastructure. It's crucial because:

- *Mapping: Maps Terraform configuration to real-world resources*
- *Performance: Caches resource attributes for faster operations*

- *Collaboration: Enables team collaboration through remote state*
- *Dependency tracking: Understands resource relationships*

Without state, Terraform wouldn't know what resources it manages or their current configuration. I always use remote state in production to enable team collaboration and prevent conflicts."

Code Example:

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

# Data source to reference another state file
data "terraform_remote_state" "network" {
  backend = "s3"
  config = {
    bucket = "my-terraform-state-bucket"
    key    = "network/terraform.tfstate"
    region = "us-west-2"
  }
}

# Using remote state data
resource "aws_instance" "app_server" {
  ami           = "ami-0c55b159cbfafe1d0"
  instance_type = "t3.micro"
  subnet_id     = data.terraform_remote_state.network.outputs.private_subnet_id
}
```

Practice Questions:

1. How do you migrate from local state to remote state?
2. What are the benefits of using DynamoDB for state locking?
3. How do you import existing infrastructure into Terraform state?

1.2 VERY IMPORTANT (Core Understanding)

Q4: Explain Terraform providers and how they work.

How to Answer in Interview:
"Providers are plugins that enable Terraform to interact with cloud platforms, SaaS providers, and other APIs. Each provider:

- *Defines resource types and data sources*

- *Handles authentication with the target platform*
- *Translates Terraform operations into API calls*
- *Manages resource lifecycle (create, read, update, delete)*

For example, the AWS provider knows how to create EC2 instances, while the Kubernetes provider manages pods and services."

Code Example:

```
# Multiple providers example
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.16"
    }
    helm = {
      source  = "hashicorp/helm"
      version = "~> 2.8"
    }
  }
}

# AWS Provider configuration
provider "aws" {
  region = "us-west-2"

  default_tags {
    tags = {
      Project     = "MyApp"
      Environment = "Production"
      ManagedBy   = "Terraform"
    }
  }
}

# Kubernetes provider (depends on EKS cluster)
provider "kubernetes" {
  host                   = aws_eks_cluster.main.endpoint
  cluster_ca_certificate = base64decode(aws_eks_cluster.main.certificate_authority[0].dat
  token                  = data.aws_eks_cluster_auth.main.token
}

# Using provider aliases for multi-region deployment
provider "aws" {
  alias  = "east"
  region = "us-east-1"
}

resource "aws_s3_bucket" "backup" {
```

```
  provider = aws.east
  bucket   = "my-backup-bucket-east"
}
```

Practice Questions:

1. How do you configure provider authentication for AWS?
2. Explain the difference between required_providers and provider blocks.
3. How do you use provider aliases for multi-region deployments?

Q5: What are Terraform modules and why are they important?

How to Answer in Interview:
"Modules are reusable components that encapsulate a set of resources. They're important because:

- *Reusability: Write once, use multiple times*
- *Organization: Logical grouping of related resources*
- *Abstraction: Hide complexity behind simple interfaces*
- *Standards: Enforce organizational best practices*
- *Testing: Easier to test smaller, focused components*

I use modules to create standardized infrastructure patterns, like a 'web-tier' module that includes load balancer, auto-scaling group, and security groups."

Code Example:

```
# Module structure
# modules/web-tier/
# ├── main.tf
# ├── variables.tf
# ├── outputs.tf
# └── README.md

# modules/web-tier/variables.tf
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t3.micro"
}

variable "min_size" {
  description = "Minimum number of instances"
  type        = number
  default     = 1
}

variable "max_size" {
  description = "Maximum number of instances"
  type        = number
  default     = 3
```

```
}

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

# modules/web-tier/main.tf
resource "aws_launch_template" "web" {
  name_prefix   = "web-tier-"
  image_id      = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  vpc_security_group_ids = [aws_security_group.web.id]

  tag_specifications {
    resource_type = "instance"
    tags = {
      Name = "WebTier"
    }
  }
}

resource "aws_autoscaling_group" "web" {
  name                = "web-tier-asg"
  vpc_zone_identifier = var.subnet_ids
  target_group_arns   = [aws_lb_target_group.web.arn]
  health_check_type   = "ELB"

  min_size         = var.min_size
  max_size         = var.max_size
  desired_capacity = var.min_size

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

# modules/web-tier/outputs.tf
output "autoscaling_group_name" {
  description = "Name of the Auto Scaling Group"
  value       = aws_autoscaling_group.web.name
}

output "load_balancer_dns" {
  description = "DNS name of the load balancer"
  value       = aws_lb.web.dns_name
}

# Using the module in root configuration
module "production_web_tier" {
  source = "./modules/web-tier"

  instance_type = "t3.medium"
  min_size      = 2
```

```
  max_size      = 10
  subnet_ids    = data.aws_subnets.private.ids
}

# Publishing module to Terraform Registry
module "web_tier" {
  source  = "myorg/web-tier/aws"
  version = "~> 1.0"

  instance_type = "t3.large"
  subnet_ids    = var.subnet_ids
}
```

Practice Questions:

1. Create a module for a VPC with public and private subnets.
2. How do you version and publish modules to the Terraform Registry?
3. Explain the difference between local and remote modules.

Category 2: State Management and Backends (HIGH PRIORITY)

2.1 VERY IMPORTANT (State Management)

Q6: How do you handle remote state and state locking?

How to Answer in Interview:
"Remote state is essential for team collaboration. I configure backends like S3 with DynamoDB for locking:

- *S3 stores the state file with versioning and encryption*
- *DynamoDB provides state locking to prevent concurrent modifications*
- *This setup ensures team members can collaborate safely*
- *State locking prevents corruption from simultaneous operations*

I always enable encryption and versioning for security and recovery."

Code Example:

```
# Backend configuration for AWS
terraform {
  backend "s3" {
    bucket         = "company-terraform-state"
    key            = "production/infrastructure.tfstate"
    region         = "us-west-2"
    dynamodb_table = "terraform-state-lock"
    encrypt        = true

    # Optional: Server-side encryption with KMS
    kms_key_id = "arn:aws:kms:us-west-2:123456789012:key/12345678-1234-1234-1234-12345678
  }
```

```
}

# S3 bucket setup for state storage
resource "aws_s3_bucket" "terraform_state" {
  bucket = "company-terraform-state"
}

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"
    }
  }
}

resource "aws_s3_bucket_public_access_block" "terraform_state" {
  bucket = aws_s3_bucket.terraform_state.id

  block_public_acls       = true
  block_public_policy     = true
  ignore_public_acls      = true
  restrict_public_buckets = true
}

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

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

  tags = {
    Name = "TerraformStateLock"
  }
}

# Alternative: Azure backend
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate"
    storage_account_name = "tfstate09762"
    container_name       = "tfstate"
    key                  = "production.terraform.tfstate"
```

```
  }
}

# State management commands
# terraform state list                    # List all resources in state
# terraform state show aws_instance.web   # Show detailed resource state
# terraform state mv aws_instance.old aws_instance.new  # Move resource in state
# terraform state rm aws_instance.example # Remove resource from state
# terraform import aws_instance.web i-1234567890abcdef0  # Import existing resource
```

Practice Questions:

1. How do you migrate state from one backend to another?
2. What happens when state locking fails and how do you resolve it?
3. How do you handle state file corruption or loss?

Q7: Explain Terraform workspaces and when to use them.

How to Answer in Interview:
"Workspaces allow you to manage multiple environments with the same configuration but separate state files. Each workspace maintains its own state, enabling:

- *Environment isolation (dev, staging, prod)*
- *Reduced code duplication*
- *Consistent configuration across environments*
- *Easy switching between environments*

However, I prefer separate directories for major environments to avoid accidental cross-environment changes."

Code Example:

```
# Using workspaces in configuration
locals {
  environment = terraform.workspace

  # Environment-specific configurations
  instance_configs = {
    dev = {
      instance_type = "t3.micro"
      min_size      = 1
      max_size      = 2
    }
    staging = {
      instance_type = "t3.small"
      min_size      = 2
      max_size      = 4
    }
    prod = {
      instance_type = "t3.medium"
      min_size      = 3
      max_size      = 10
```

```
    }
  }

  config = local.instance_configs[local.environment]
}

# Resources using workspace-aware configuration
resource "aws_instance" "app_server" {
  count         = local.config.min_size
  ami           = data.aws_ami.ubuntu.id
  instance_type = local.config.instance_type

  tags = {
    Name        = "AppServer-${local.environment}-${count.index + 1}"
    Environment = local.environment
    Workspace   = terraform.workspace
  }
}

# S3 bucket with workspace-specific naming
resource "aws_s3_bucket" "app_data" {
  bucket = "myapp-data-${local.environment}-${random_id.bucket_suffix.hex}"
}

# Workspace commands examples:
# terraform workspace list              # List all workspaces
# terraform workspace new development   # Create new workspace
# terraform workspace select production # Switch to workspace
# terraform workspace show             # Show current workspace
# terraform workspace delete staging   # Delete workspace

# Variable files for different workspaces
# variables/dev.tfvars
# environment = "development"
# instance_type = "t3.micro"

# variables/prod.tfvars
# environment = "production"
# instance_type = "t3.large"

# Apply with specific variable file
# terraform apply -var-file="variables/prod.tfvars"
```

Practice Questions:

1. Compare workspaces vs. separate directories for environment management.
2. How do you handle workspace-specific variables and configurations?
3. What are the limitations of using Terraform workspaces?

2.2 IMPORTANT (Advanced State Operations)

Q8: How do you import existing infrastructure into Terraform?

How to Answer in Interview:
"Importing existing infrastructure involves two steps:

1. *Write the Terraform configuration for the existing resource*
2. *Use terraform import to add it to state*

This is useful when adopting Terraform for existing infrastructure or when resources were created outside Terraform. I always verify the configuration matches the existing resource exactly."

Code Example:

```
# Step 1: Write configuration for existing resource
resource "aws_instance" "existing_server" {
  ami           = "ami-0c55b159cbfafe1d0"  # Must match existing
  instance_type = "t3.medium"             # Must match existing

  # Include all current configuration
  tags = {
    Name = "ExistingServer"
  }
}

# Step 2: Import the resource
# terraform import aws_instance.existing_server i-1234567890abcdef0

# For complex resources, use import blocks (Terraform 1.5+)
import {
  to = aws_instance.existing_server
  id = "i-1234567890abcdef0"
}

# Batch import script example
#!/bin/bash
# import_infrastructure.sh

# Import VPC
terraform import aws_vpc.main vpc-12345678

# Import subnets
terraform import aws_subnet.public_1 subnet-12345678
terraform import aws_subnet.public_2 subnet-87654321

# Import security groups
terraform import aws_security_group.web sg-12345678

# Import instances
terraform import aws_instance.web_1 i-1234567890abcdef0
terraform import aws_instance.web_2 i-0987654321fedcba0
```

```
# Verify imports
terraform plan

# Example: Import entire AWS infrastructure
# Use tools like terraformer
# terraformer import aws --resources=vpc,subnet,sg,ec2_instance --regions=us-west-2

# Generated configuration cleanup
resource "aws_instance" "web_server" {
  # Remove unnecessary computed attributes
  ami                     = "ami-0c55b159cbfafe1d0"
  instance_type          = "t3.medium"
  key_name               = "my-key-pair"
  vpc_security_group_ids = [aws_security_group.web.id]
  subnet_id              = aws_subnet.public.id

  # Keep only essential attributes
  tags = {
    Name = "WebServer"
  }

  # Remove:
  # - arn
  # - id
  # - public_ip
  # - private_ip
  # - etc. (computed values)
}
```

Practice Questions:

1. How do you handle importing resources with complex dependencies?
2. What tools can help automate the import process for large infrastructures?
3. How do you validate imported resources match your configuration?

Category 3: Advanced Terraform Features (MEDIUM-HIGH PRIORITY)

3.1 CONFIGURATION MANAGEMENT

Q9: Explain variables, locals, and outputs in Terraform.

How to Answer in Interview:
"These are key components for making Terraform configurations flexible and reusable:

- *Variables: Input parameters that make configurations customizable*
- *Locals: Computed values for internal use, reduce repetition*
- *Outputs: Return values that can be used by other configurations*

I use variables for user inputs, locals for computed values and complex expressions, and outputs to share data between modules or configurations."

Code Example:

```
# Variable definitions (variables.tf)
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 "instance_config" {
  description = "Instance configuration"
  type = object({
    instance_type = string
    min_size     = number
    max_size     = number
  })
  default = {
    instance_type = "t3.micro"
    min_size     = 1
    max_size     = 3
  }
}

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

# Local values (locals.tf)
locals {
  # Computed naming convention
  name_prefix = "${var.environment}-myapp"

  # Common tags merged with environment-specific tags
  common_tags = merge(var.tags, {
    Environment = var.environment
    Project     = "MyApplication"
    ManagedBy   = "Terraform"
    Timestamp   = timestamp()
  })

  # Environment-specific configurations
  environment_configs = {
    dev = {
      instance_type = "t3.micro"
      min_size      = 1
      max_size      = 2
      enable_backup = false
    }
    staging = {
      instance_type = "t3.small"
      min_size      = 2
      max_size      = 4
      enable_backup = true
```

```
    }
    prod = {
      instance_type = "t3.medium"
      min_size      = 3
      max_size      = 10
      enable_backup = true
    }
  }

  config = local.environment_configs[var.environment]

  # Complex expressions
  subnet_cidrs = [
    for i in range(length(data.aws_availability_zones.available.names)) :
    cidrsubnet("10.0.0.0/16", 8, i)
  ]
}

# Resource using variables and locals
resource "aws_instance" "app_server" {
  count         = local.config.min_size
  ami           = data.aws_ami.ubuntu.id
  instance_type = local.config.instance_type

  tags = merge(local.common_tags, {
    Name = "${local.name_prefix}-app-${count.index + 1}"
    Role = "application"
  })
}

# Outputs (outputs.tf)
output "instance_ids" {
  description = "IDs of the created instances"
  value       = aws_instance.app_server[*].id
}

output "instance_ips" {
  description = "Public IP addresses"
  value = {
    public  = aws_instance.app_server[*].public_ip
    private = aws_instance.app_server[*].private_ip
  }
}

output "load_balancer_url" {
  description = "Load balancer URL"
  value       = "https://${aws_lb.app.dns_name}"
  sensitive   = false
}

output "database_endpoint" {
  description = "RDS endpoint"
  value       = aws_db_instance.main.endpoint
  sensitive   = true  # Mark sensitive data
}
```

```
# Using outputs from modules
module "vpc" {
  source = "./modules/vpc"

  environment = var.environment
}

module "app_tier" {
  source = "./modules/app-tier"

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnet_ids
}

# Variable files (terraform.tfvars)
environment = "production"

instance_config = {
  instance_type = "t3.large"
  min_size     = 5
  max_size     = 20
}

tags = {
  CostCenter = "Engineering"
  Owner      = "Platform Team"
}

# Environment-specific variable files
# environments/dev.tfvars
# environments/staging.tfvars
# environments/prod.tfvars
```

Practice Questions:

1. How do you implement variable validation and default values?
2. When should you use locals vs. variables?
3. How do you handle sensitive outputs and variables?

Q10: Explain conditional expressions and loops in Terraform.

How to Answer in Interview:
"Terraform provides several ways to handle conditional logic and iteration:

- *Conditional expressions: condition ? true_val : false_val*
- *For expressions: for item in list : transformation*
- *Count parameter: Create multiple resources*
- *For_each: Create resources for each item in map/set*

These features help create dynamic and flexible configurations that adapt to different requirements."

Code Example:

```
# Conditional expressions
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.environment == "prod" ? "t3.large" : "t3.micro"

  # Conditional resource creation
  count = var.create_instance ? 1 : 0

  tags = {
    Name = "WebServer"
    Backup = var.environment == "prod" ? "daily" : "weekly"
  }
}

# For expressions
locals {
  # Transform list to map
  availability_zones = ["us-west-2a", "us-west-2b", "us-west-2c"]

  az_map = {
    for az in local.availability_zones :
    az => "${az}-subnet"
  }

  # Filter and transform
  production_instances = {
    for name, config in var.instances :
    name => config
    if config.environment == "production"
  }

  # Complex transformations
  subnet_configs = [
    for i, az in local.availability_zones : {
      name              = "subnet-${i + 1}"
      availability_zone = az
      cidr_block       = cidrsubnet("10.0.0.0/16", 8, i)
      public           = i < 2  # First two are public
    }
  ]
}

# Count parameter for multiple resources
resource "aws_subnet" "public" {
  count = length(local.availability_zones)

  vpc_id            = aws_vpc.main.id
  cidr_block        = cidrsubnet("10.0.0.0/16", 8, count.index)
  availability_zone = local.availability_zones[count.index]

  map_public_ip_on_launch = true

  tags = {
    Name = "PublicSubnet-${count.index + 1}"
    Type = "public"
  }
```

```
}

# For_each with maps
variable "users" {
  type = map(object({
    role   = string
    groups = list(string)
  }))
  default = {
    alice = {
      role   = "admin"
      groups = ["admins", "developers"]
    }
    bob = {
      role   = "developer"
      groups = ["developers"]
    }
  }
}

resource "aws_iam_user" "users" {
  for_each = var.users

  name = each.key

  tags = {
    Role = each.value.role
  }
}

resource "aws_iam_user_group_membership" "user_groups" {
  for_each = var.users

  user   = aws_iam_user.users[each.key].name
  groups = each.value.groups
}

# For_each with sets
variable "security_group_rules" {
  type = set(object({
    port        = number
    protocol    = string
    cidr_blocks = list(string)
  }))
  default = [
    {
      port        = 80
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    },
    {
      port        = 443
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  ]
```

```
}

resource "aws_security_group_rule" "ingress" {
  for_each = {
    for rule in var.security_group_rules :
    "${rule.protocol}-${rule.port}" => rule
  }

  type              = "ingress"
  from_port         = each.value.port
  to_port           = each.value.port
  protocol          = each.value.protocol
  cidr_blocks       = each.value.cidr_blocks
  security_group_id = aws_security_group.web.id
}

# Dynamic blocks
resource "aws_security_group" "web" {
  name_prefix = "web-"
  vpc_id      = aws_vpc.main.id

  # Dynamic ingress rules
  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }

  # Multiple dynamic blocks
  dynamic "egress" {
    for_each = var.egress_rules
    content {
      from_port   = egress.value.port
      to_port     = egress.value.port
      protocol    = egress.value.protocol
      cidr_blocks = egress.value.cidr_blocks
    }
  }
}

# Splat expressions
locals {
  # Extract specific attributes from resources
  instance_ids   = aws_instance.web[*].id
  private_ips    = aws_instance.web[*].private_ip

  # Work with maps
  user_arns = values(aws_iam_user.users)[*].arn
}
```

Practice Questions:

1. How do you choose between count and for_each?
2. Implement a module that creates IAM users with different permissions.
3. How do you handle complex nested data structures with for expressions?

3.2 BEST PRACTICES AND OPTIMIZATION

Q11: What are Terraform best practices for production use?

How to Answer in Interview:
"Production Terraform requires careful planning and best practices:

- *State management: Use remote state with locking*
- *Code organization: Separate environments, use modules*
- *Security: Encrypt state, use IAM roles, scan for secrets*
- *CI/CD integration: Automate plan and apply processes*
- *Testing: Validate configurations before applying*
- *Documentation: Clear README and variable descriptions*

I always follow the principle of least privilege and implement proper code review processes."

Code Example:

```
# Project structure best practices
project/
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── terraform.tfvars
│   │   └── backend.tf
│   ├── staging/
│   └── prod/
├── modules/
│   ├── vpc/
│   ├── compute/
│   └── database/
├── policies/
│   └── terraform-policy.json
└── scripts/
    ├── plan.sh
    └── apply.sh

# Security best practices
# Use data sources for sensitive information
data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "prod/database/password"
}

# IAM role for Terraform execution
resource "aws_iam_role" "terraform_role" {
  name = "terraform-execution-role"
```

```
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action = "sts:AssumeRole"
        Effect = "Allow"
        Principal = {
          AWS = "arn:aws:iam::${data.aws_caller_identity.current.account_id}:root"
        }
        Condition = {
          StringEquals = {
            "sts:ExternalId" = var.external_id
          }
        }
      }
    ]
  })
}

# Least privilege policy
resource "aws_iam_role_policy" "terraform_policy" {
  name = "terraform-policy"
  role = aws_iam_role.terraform_role.id

  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Effect = "Allow"
        Action = [
          "ec2:*",
          "s3:*",
          "iam:Get*",
          "iam:List*"
        ]
        Resource = "*"
      }
    ]
  })
}

# Resource tagging standards
locals {
  required_tags = {
    Environment   = var.environment
    Project       = var.project_name
    Owner         = var.owner
    CostCenter    = var.cost_center
    ManagedBy     = "terraform"
    CreatedDate   = formatdate("YYYY-MM-DD", timestamp())
  }
}

# Naming conventions
locals {
```

```
  naming_prefix = "${var.project_name}-${var.environment}"

  # Consistent naming
  vpc_name = "${local.naming_prefix}-vpc"
  sg_name  = "${local.naming_prefix}-sg"
}

# Resource lifecycle management
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  # Prevent accidental destruction
  lifecycle {
    prevent_destroy = true

    # Ignore changes to ami (managed by auto-update)
    ignore_changes = [ami]

    # Create before destroy for zero downtime
    create_before_destroy = true
  }

  tags = local.required_tags
}

# Data validation
variable "environment" {
  description = "Environment name"
  type        = string

  validation {
    condition     = can(regex("^(dev|staging|prod)$", var.environment))
    error_message = "Environment must be dev, staging, or prod."
  }
}

# CI/CD Pipeline configuration (.github/workflows/terraform.yml)
```yaml
name: Terraform

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

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v2
```
```

```yaml
      with:
        terraform_version: 1.5.0

    - name: Terraform Init
      run: terraform init

    - name: Terraform Validate
      run: terraform validate

    - name: Terraform Plan
      run: terraform plan -no-color

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

Makefile for common operations

```makefile
.PHONY: init plan apply destroy validate fmt lint

init:
	terraform init

validate:
	terraform validate

fmt:
	terraform fmt -recursive

lint:
	tflint

plan:
	terraform plan -var-file="environments/$(ENV).tfvars"

apply:
	terraform apply -var-file="environments/$(ENV).tfvars"

destroy:
	terraform destroy -var-file="environments/$(ENV).tfvars"

# Usage: make plan ENV=dev
```

Practice Questions:

1. How do you implement automated testing for Terraform configurations?
2. What security scanning tools do you use for Terraform code?
3. How do you handle secrets and sensitive data in Terraform?

Category 4: Advanced Scenarios and Troubleshooting (MEDIUM PRIORITY)

4.1 TROUBLESHOOTING AND DEBUGGING

Q12: How do you troubleshoot common Terraform issues?

How to Answer in Interview:
"I follow a systematic approach to troubleshooting:

1. *Check terraform validate for syntax errors*
2. *Review terraform plan output carefully*
3. *Use TF_LOG for detailed debugging*
4. *Check provider versions and compatibility*
5. *Verify state file integrity*
6. *Review cloud provider logs and limits*

Common issues include state drift, resource dependencies, and permission problems. I always start with the simplest checks first."

Code Example:

```
# Debugging environment variables
export TF_LOG=DEBUG
export TF_LOG_PATH=./terraform.log

# Common troubleshooting commands
terraform validate                    # Check syntax
terraform plan -detailed-exitcode    # Check for changes
terraform refresh                     # Update state with real infrastructure
terraform state list                 # List all resources
terraform state show aws_instance.web # Show specific resource

# State file issues
terraform state pull > state.backup  # Backup state
terraform force-unlock LOCK_ID       # Unlock stuck state
terraform state rm aws_instance.web  # Remove from state
terraform import aws_instance.web i-12345 # Re-import resource

# Common error scenarios and solutions

# 1. Resource already exists error
# Error: resource already exists
# Solution: Import existing resource or remove from state
terraform import aws_s3_bucket.example my-existing-bucket

# 2. Dependency cycle error
# Error: Cycle detected in resource dependencies
# Solution: Use depends_on or refactor resources
resource "aws_instance" "web" {
  # ... configuration
  depends_on = [aws_security_group.web]
```

```
}

# 3. Provider version conflicts
# Error: provider version incompatible
# Solution: Update provider constraints
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

# 4. State drift detection and resolution
# Check for drift
terraform plan -detailed-exitcode
if [ $? -eq 2 ]; then
  echo "Configuration drift detected"
  terraform plan -out=plan.tfplan
  # Review plan before applying
  terraform show plan.tfplan
fi

# 5. Permission issues debugging
# AWS CLI test
aws sts get-caller-identity
aws ec2 describe-instances --region us-west-2

# 6. Resource timeout issues
resource "aws_db_instance" "main" {
  # ... configuration

  timeouts {
    create = "40m"
    delete = "40m"
    update = "80m"
  }
}

# 7. Memory and performance issues
# Use smaller plan/apply operations
terraform plan -target=aws_instance.web
terraform apply -target=aws_instance.web

# 8. Module debugging
# Check module source
terraform get -update
terraform init -upgrade

# Validation script for common issues
#!/bin/bash
# validate_terraform.sh

echo "Checking Terraform version..."
terraform version
```

```
echo "Validating configuration..."
terraform validate || exit 1

echo "Checking formatting..."
terraform fmt -check || {
  echo "Code not formatted. Run: terraform fmt"
  exit 1
}

echo "Running security scan..."
tfsec . || echo "Security issues found"

echo "Checking for state drift..."
terraform plan -detailed-exitcode
case $? in
  0) echo "No changes needed" ;;
  1) echo "Terraform plan failed"; exit 1 ;;
  2) echo "Changes detected in plan" ;;
esac

echo "All checks passed!"
```

Practice Questions:

1. How do you recover from a corrupted state file?
2. What steps do you take when terraform apply fails halfway through?
3. How do you debug complex module dependency issues?

4.2 ADVANCED TERRAFORM PATTERNS

Q13: Explain advanced Terraform patterns and when to use them.

Code Example:

```
# 1. Data-driven infrastructure
# Dynamically create resources based on data
variable "environments" {
  type = map(object({
    vpc_cidr         = string
    instance_type    = string
    min_size        = number
    max_size        = number
    enable_monitoring = bool
  }))

  default = {
    dev = {
      vpc_cidr         = "10.0.0.0/16"
      instance_type    = "t3.micro"
      min_size        = 1
      max_size        = 2
      enable_monitoring = false
```

```
    }
    prod = {
      vpc_cidr         = "10.1.0.0/16"
      instance_type    = "t3.large"
      min_size        = 3
      max_size        = 10
      enable_monitoring = true
    }
  }
}

# Create environments dynamically
module "environments" {
  source = "./modules/environment"

  for_each = var.environments

  environment_name = each.key
  vpc_cidr        = each.value.vpc_cidr
  instance_type   = each.value.instance_type
  min_size       = each.value.min_size
  max_size       = each.value.max_size

  # Conditional resources
  monitoring_enabled = each.value.enable_monitoring
}

# 2. Factory pattern for similar resources
module "application_stack" {
  source = "./modules/app-factory"

  for_each = toset(["api", "web", "worker"])

  application_name = each.key
  environment     = var.environment

  # App-specific configurations
  config = {
    api = {
      instance_type = "t3.medium"
      port         = 8080
      health_check = "/health"
    }
    web = {
      instance_type = "t3.small"
      port         = 80
      health_check = "/"
    }
    worker = {
      instance_type = "t3.large"
      port         = null
      health_check = null
    }
  }[each.key]
}
```

```
# 3. Blue-Green deployment pattern
resource "aws_launch_template" "app" {
  name_prefix = "${var.app_name}-${var.environment}-"

  image_id      = var.ami_id
  instance_type = var.instance_type

  # Use timestamp to force new launch template
  tag_specifications {
    resource_type = "instance"
    tags = {
      Name    = "${var.app_name}-${var.environment}"
      Version = var.app_version
      Color   = var.deployment_color  # blue or green
    }
  }

  lifecycle {
    create_before_destroy = true
  }
}

resource "aws_autoscaling_group" "app" {
  name                = "${var.app_name}-${var.environment}-${var.deployment_color}"
  vpc_zone_identifier = var.subnet_ids
  target_group_arns   = [aws_lb_target_group.app.arn]

  min_size         = var.min_size
  max_size         = var.max_size
  desired_capacity = var.desired_capacity

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

  # Lifecycle hooks for graceful deployment
  initial_lifecycle_hook {
    name         = "instance-launch"
    lifecycle_transition = "autoscaling:EC2_INSTANCE_LAUNCHING"
    default_result = "ABANDON"
    heartbeat_timeout = 300
  }

  tag {
    key                 = "Color"
    value               = var.deployment_color
    propagate_at_launch = false
  }
}

# 4. Progressive resource creation
resource "aws_vpc" "main" {
  cidr_block = var.vpc_cidr

  # Enable features based on environment
```

```
  enable_dns_hostnames = var.environment == "prod" ? true : false
  enable_dns_support   = true

  tags = local.common_tags
}

# Create subnets progressively
resource "aws_subnet" "private" {
  count = var.environment == "prod" ? 3 : 1

  vpc_id            = aws_vpc.main.id
  cidr_block        = cidrsubnet(var.vpc_cidr, 8, count.index + 10)
  availability_zone = data.aws_availability_zones.available.names[count.index]

  tags = merge(local.common_tags, {
    Name = "PrivateSubnet-${count.index + 1}"
    Tier = "private"
  })
}

# 5. Resource composition pattern
module "database_cluster" {
  source = "./modules/database"

  # Conditional database creation
  create_database = var.environment != "dev"

  # Use different configs per environment
  engine_version = var.environment == "prod" ? "13.7" : "13.4"
  instance_class = var.environment == "prod" ? "db.r5.large" : "db.t3.micro"

  # Multi-AZ only in production
  multi_az = var.environment == "prod"

  # Backup configuration
  backup_retention_period = var.environment == "prod" ? 30 : 7
  backup_window          = var.environment == "prod" ? "03:00-04:00" : "06:00-07:00"
}

# 6. Template rendering pattern
data "template_file" "user_data" {
  template = file("${path.module}/scripts/user_data.sh.tpl")

  vars = {
    app_name    = var.app_name
    environment = var.environment
    region      = data.aws_region.current.name

    # Environment-specific configurations
    log_level = var.environment == "prod" ? "ERROR" : "DEBUG"

    # Service discovery endpoints
    database_endpoint = module.database_cluster.endpoint
    cache_endpoint   = aws_elasticache_cluster.main.cache_nodes[0].address
  }
}
```

```
resource "aws_instance" "app" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type
  user_data     = data.template_file.user_data.rendered

  tags = local.common_tags
}
```

Practice Questions:

1. Design a multi-tenant infrastructure using Terraform.
2. Implement a canary deployment strategy with Terraform.
3. How do you handle complex inter-module dependencies?

Category 5: Integration and Ecosystem (MEDIUM PRIORITY)

5.1 CI/CD INTEGRATION

Q14: How do you integrate Terraform with CI/CD pipelines?

How to Answer in Interview:
"I integrate Terraform into CI/CD pipelines with these principles:

- *Separate plan and apply stages for review*
- *Use pull request workflows for plan validation*
- *Implement automated testing and security scanning*
- *Store state remotely with proper access controls*
- *Use service accounts with minimal permissions*

This ensures infrastructure changes are reviewed, tested, and deployed consistently."

Code Example:

```
# GitLab CI pipeline (.gitlab-ci.yml)
stages:
  - validate
  - plan
  - apply
  - destroy

variables:
  TF_ROOT: ${CI_PROJECT_DIR}/terraform
  TF_ADDRESS: ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/terraform/state/${CI_COMMIT_REF_

before_script:
  - cd ${TF_ROOT}
  - terraform --version
  - terraform init
```

```yaml
validate:
  stage: validate
  script:
    - terraform validate
    - terraform fmt -check
  only:
    - merge_requests
    - main

plan:
  stage: plan
  script:
    - terraform plan -out="planfile"
    - terraform show -json planfile > plan.json
  artifacts:
    paths:
      - ${TF_ROOT}/planfile
      - ${TF_ROOT}/plan.json
    expire_in: 1 week
  only:
    - merge_requests
    - main

apply:
  stage: apply
  script:
    - terraform apply -input=false "planfile"
  dependencies:
    - plan
  only:
    - main
  when: manual

# GitHub Actions workflow (.github/workflows/terraform.yml)
name: Terraform

on:
  push:
    branches: [main]
    paths: ['terraform/**']
  pull_request:
    branches: [main]
    paths: ['terraform/**']

jobs:
  terraform:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: terraform

    permissions:
      contents: read
      pull-requests: write
      id-token: write  # For OIDC authentication
```

```yaml
    steps:
    - name: Checkout
      uses: actions/checkout@v3

    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v2
      with:
        role-to-assume: ${{ secrets.AWS_ROLE_ARN }}
        aws-region: us-west-2

    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v2
      with:
        terraform_version: 1.5.0

    - name: Terraform Init
      run: terraform init

    - name: Terraform Validate
      run: terraform validate

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

    - name: Run Security Scan
      uses: aquasecurity/tfsec-action@v1.0.0
      with:
        working_directory: terraform

    - name: Terraform Plan
      id: plan
      run: |
        terraform plan -no-color -out=tfplan
        terraform show -no-color tfplan > plan_output.txt
      continue-on-error: true

    - name: Comment PR
      uses: actions/github-script@v6
      if: github.event_name == 'pull_request'
      with:
        script: |
          const fs = require('fs');
          const plan = fs.readFileSync('terraform/plan_output.txt', 'utf8');
          const maxGitHubBodyCharacters = 65536;

          function chunkSubstr(str, size) {
            const numChunks = Math.ceil(str.length / size)
            const chunks = new Array(numChunks)
            for (let i = 0, o = 0; i < numChunks; ++i, o += size) {
              chunks[i] = str.substr(o, size)
            }
            return chunks
          }

          const planChunks = chunkSubstr(plan, maxGitHubBodyCharacters);
```

```
          for (let i = 0; i < planChunks.length; i++) {
            const output = `#### Terraform Plan Output (Part ${i + 1}/${planChunks.length}
            \`\`\`
            ${planChunks[i]}
            \`\`\``;

            await github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: output
            });
          }

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

# Azure DevOps pipeline (azure-pipelines.yml)
trigger:
  branches:
    include:
      - main
  paths:
    include:
      - terraform/*

pool:
  vmImage: 'ubuntu-latest'

variables:
  - group: terraform-variables
  - name: workingDirectory
    value: '$(System.DefaultWorkingDirectory)/terraform'

stages:
- stage: Validate
  jobs:
  - job: validate
    displayName: 'Validate Terraform'
    steps:
    - task: TerraformInstaller@0
      displayName: 'Install Terraform'
      inputs:
        terraformVersion: '1.5.0'

    - task: TerraformTaskV2@2
      displayName: 'Terraform Init'
      inputs:
        provider: 'azurerm'
        command: 'init'
        workingDirectory: '$(workingDirectory)'
        backendServiceArm: 'terraform-backend'
        backendAzureRmResourceGroupName: 'terraform-state-rg'
        backendAzureRmStorageAccountName: 'tfstatestorage'
        backendAzureRmContainerName: 'tfstate'
```

```
        backendAzureRmKey: 'terraform.tfstate'

    - task: TerraformTaskV2@2
      displayName: 'Terraform Validate'
      inputs:
        provider: 'azurerm'
        command: 'validate'
        workingDirectory: '$(workingDirectory)'

- stage: Plan
  dependsOn: Validate
  condition: succeeded()
  jobs:
  - job: plan
    displayName: 'Plan Terraform'
    steps:
    - task: TerraformTaskV2@2
      displayName: 'Terraform Plan'
      inputs:
        provider: 'azurerm'
        command: 'plan'
        workingDirectory: '$(workingDirectory)'
        environmentServiceNameAzureRM: 'terraform-backend'

- stage: Apply
  dependsOn: Plan
  condition: and(succeeded(), eq(variables['Build.SourceBranch'], 'refs/heads/main'))
  jobs:
  - deployment: apply
    displayName: 'Apply Terraform'
    environment: production
    strategy:
      runOnce:
        deploy:
          steps:
          - task: TerraformTaskV2@2
            displayName: 'Terraform Apply'
            inputs:
              provider: 'azurerm'
              command: 'apply'
              workingDirectory: '$(workingDirectory)'
              environmentServiceNameAzureRM: 'terraform-backend'

# Terragrunt with CI/CD
# terragrunt.hcl
terraform {
  source = "git::https://github.com/company/terraform-modules.git//vpc?ref=v1.0.0"
}

include {
  path = find_in_parent_folders()
}

inputs = {
  environment = "production"
  vpc_cidr    = "10.0.0.0/16"
```

```
}

# CI/CD script with Terragrunt
#!/bin/bash
# deploy.sh

set -e

ENVIRONMENT=${1:-dev}
ACTION=${2:-plan}

echo "Running Terragrunt ${ACTION} for ${ENVIRONMENT}"

cd environments/${ENVIRONMENT}

case ${ACTION} in
  init)
    terragrunt init
    ;;
  plan)
    terragrunt plan
    ;;
  apply)
    terragrunt apply -auto-approve
    ;;
  destroy)
    terragrunt destroy -auto-approve
    ;;
  *)
    echo "Unknown action: ${ACTION}"
    exit 1
    ;;
esac
```

Practice Questions:

1. How do you handle Terraform state in a multi-branch CI/CD workflow?
2. Implement automated testing for Terraform modules in CI/CD.
3. How do you manage Terraform deployments across multiple environments?

5.2 MONITORING AND COMPLIANCE

Q15: How do you implement monitoring and compliance for Terraform-managed infrastructure?

Code Example:

```
# Compliance and monitoring configuration

# 1. AWS Config for compliance monitoring
resource "aws_config_configuration_recorder" "main" {
  name     = "terraform-compliance-recorder"
  role_arn = aws_iam_role.config.arn
```

```
  recording_group {
    all_supported                 = true
    include_global_resource_types = true
  }
}

resource "aws_config_delivery_channel" "main" {
  name           = "terraform-compliance-channel"
  s3_bucket_name = aws_s3_bucket.config.bucket

  snapshot_delivery_properties {
    delivery_frequency = "Daily"
  }
}

# Compliance rules
resource "aws_config_config_rule" "s3_encrypted" {
  name = "s3-bucket-server-side-encryption-enabled"

  source {
    owner             = "AWS"
    source_identifier = "S3_BUCKET_SERVER_SIDE_ENCRYPTION_ENABLED"
  }

  depends_on = [aws_config_configuration_recorder.main]
}

resource "aws_config_config_rule" "security_groups" {
  name = "security-group-ssh-check"

  source {
    owner             = "AWS"
    source_identifier = "INCOMING_SSH_DISABLED"
  }

  depends_on = [aws_config_configuration_recorder.main]
}

# 2. CloudWatch monitoring
resource "aws_cloudwatch_dashboard" "infrastructure" {
  dashboard_name = "terraform-infrastructure"

  dashboard_body = jsonencode({
    widgets = [
      {
        type   = "metric"
        x      = 0
        y      = 0
        width  = 12
        height = 6

        properties = {
          metrics = [
            ["AWS/EC2", "CPUUtilization", "InstanceId", aws_instance.web.id],
            ["AWS/ApplicationELB", "RequestCount", "LoadBalancer", aws_lb.main.arn_suffi
```

```
          ]
          period = 300
          stat   = "Average"
          region = "us-west-2"
          title  = "Infrastructure Metrics"
        }
      }
    ]
  })
}

# CloudWatch alarms
resource "aws_cloudwatch_metric_alarm" "high_cpu" {
  alarm_name          = "terraform-high-cpu"
  comparison_operator = "GreaterThanThreshold"
  evaluation_periods  = "2"
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = "300"
  statistic           = "Average"
  threshold           = "80"
  alarm_description   = "This metric monitors ec2 cpu utilization"
  alarm_actions       = [aws_sns_topic.alerts.arn]

  dimensions = {
    InstanceId = aws_instance.web.id
  }
}

# 3. Cost monitoring
resource "aws_budgets_budget" "infrastructure_cost" {
  name     = "terraform-infrastructure-budget"
  budget_type = "COST"
  limit_amount = "100"
  limit_unit   = "USD"
  time_unit    = "MONTHLY"

  time_period_start = "2023-01-01_00:00"

  cost_filters = {
    Tag = ["ManagedBy:Terraform"]
  }

  notification {
    comparison_operator        = "GREATER_THAN"
    threshold                 = 80
    threshold_type            = "PERCENTAGE"
    notification_type         = "ACTUAL"
    subscriber_email_addresses = ["admin@company.com"]
  }
}

# 4. Security scanning integration
resource "aws_inspector_assessment_template" "security_scan" {
  name       = "terraform-security-assessment"
  target_arn = aws_inspector_assessment_target.security.arn
```

```
  duration   = 3600

  rules_package_arns = [
    "arn:aws:inspector:us-west-2:758058086616:rulespackage/0-9hgA516p",  # Security Best
    "arn:aws:inspector:us-west-2:758058086616:rulespackage/0-H5hpSawc",  # Network Reacha
  ]

  tags = {
    ManagedBy = "Terraform"
  }
}

# 5. Drift detection
resource "aws_lambda_function" "drift_detection" {
  filename         = "drift_detection.zip"
  function_name    = "terraform-drift-detection"
  role            = aws_iam_role.lambda.arn
  handler         = "index.handler"
  source_code_hash = data.archive_file.drift_detection.output_base64sha256
  runtime         = "python3.9"
  timeout         = 300

  environment {
    variables = {
      STATE_BUCKET = aws_s3_bucket.terraform_state.bucket
      SNS_TOPIC    = aws_sns_topic.alerts.arn
    }
  }
}

# CloudWatch event to trigger drift detection
resource "aws_cloudwatch_event_rule" "drift_detection" {
  name                = "terraform-drift-detection"
  description         = "Trigger drift detection daily"
  schedule_expression = "rate(1 day)"
}

resource "aws_cloudwatch_event_target" "drift_detection" {
  rule      = aws_cloudwatch_event_rule.drift_detection.name
  target_id = "TerraformDriftDetectionTarget"
  arn       = aws_lambda_function.drift_detection.arn
}

# Policy document for security scanning
data "aws_iam_policy_document" "security_scan" {
  statement {
    effect = "Allow"

    actions = [
      "ec2:Describe*",
      "inspector:*",
      "iam:Get*",
      "iam:List*"
    ]

    resources = ["*"]
```

```
  }
}

# 6. Compliance reporting
resource "aws_s3_bucket" "compliance_reports" {
  bucket = "terraform-compliance-reports-${random_id.bucket.hex}"
}

resource "aws_lambda_function" "compliance_report" {
  filename         = "compliance_report.zip"
  function_name    = "terraform-compliance-report"
  role            = aws_iam_role.lambda.arn
  handler         = "index.handler"
  source_code_hash = data.archive_file.compliance_report.output_base64sha256
  runtime         = "python3.9"
  timeout         = 900

  environment {
    variables = {
      REPORT_BUCKET = aws_s3_bucket.compliance_reports.bucket
      CONFIG_BUCKET = aws_s3_bucket.config.bucket
    }
  }
}

# 7. Resource tagging for governance
locals {
  compliance_tags = {
    DataClassification = var.data_classification
    Compliance         = var.compliance_framework
    BackupRequired     = var.backup_required
    MonitoringLevel    = var.monitoring_level
  }
}

# Apply compliance tags to all resources
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  tags = merge(local.common_tags, local.compliance_tags, {
    Name = "WebServer"
  })
}

# 8. Automated remediation
resource "aws_lambda_function" "auto_remediation" {
  filename         = "auto_remediation.zip"
  function_name    = "terraform-auto-remediation"
  role            = aws_iam_role.lambda.arn
  handler         = "index.handler"
  source_code_hash = data.archive_file.auto_remediation.output_base64sha256
  runtime         = "python3.9"
  timeout         = 300

  environment {
```

```
    variables = {
      SNS_TOPIC = aws_sns_topic.alerts.arn
    }
  }
}

# EventBridge rule for compliance violations
resource "aws_cloudwatch_event_rule" "compliance_violation" {
  name        = "terraform-compliance-violation"
  description = "Capture compliance violations"

  event_pattern = jsonencode({
    source      = ["aws.config"]
    detail-type = ["Config Rules Compliance Change"]
    detail = {
      newEvaluationResult = {
        complianceType = ["NON_COMPLIANT"]
      }
    }
  })
}

resource "aws_cloudwatch_event_target" "auto_remediation" {
  rule      = aws_cloudwatch_event_rule.compliance_violation.name
  target_id = "AutoRemediationTarget"
  arn       = aws_lambda_function.auto_remediation.arn
}
```

Practice Questions:

1. Design a comprehensive monitoring strategy for Terraform-managed infrastructure.
2. How do you implement automated compliance checking in your Terraform workflow?
3. Create a disaster recovery plan for Terraform-managed resources.

PRACTICAL SCENARIOS AND EXERCISES

Beginner Level Practice:

1. **Basic Infrastructure**: Create a VPC with public/private subnets, EC2 instance, and security groups
2. **State Management**: Set up remote state with S3 and DynamoDB
3. **Variables and Outputs**: Create a reusable module with proper variable validation
4. **Resource Dependencies**: Build infrastructure with complex dependencies
5. **Import Exercise**: Import existing AWS resources into Terraform state

Intermediate Level Practice:

1. **Multi-Environment Setup**: Design infrastructure for dev/staging/prod environments
2. **Module Development**: Create and publish a complex module to Terraform Registry
3. **CI/CD Integration**: Implement complete GitLab/GitHub Actions pipeline
4. **State Manipulation**: Practice state import, move, and remove operations
5. **Dynamic Infrastructure**: Use for_each and count for dynamic resource creation

Advanced Level Practice:

1. **Blue-Green Deployment**: Implement automated blue-green deployment with Terraform
2. **Multi-Cloud Setup**: Deploy infrastructure across AWS, Azure, and GCP
3. **Custom Provider**: Develop a custom Terraform provider
4. **Complex Modules**: Build hierarchical modules with complex dependencies
5. **Enterprise Patterns**: Implement governance, compliance, and security patterns

INTERVIEW PREPARATION STRATEGY

How to Structure Your Terraform Answers:

1. **Start with the concept** - Define what you're explaining
2. **Explain the why** - Business or technical benefits
3. **Provide practical examples** - Show real-world usage
4. **Discuss best practices** - Demonstrate professional experience
5. **Mention alternatives** - Show broader understanding

Essential Terraform Interview Topics:

- **Core Concepts**: Resources, providers, state, modules
- **State Management**: Remote state, locking, import/export
- **Configuration**: Variables, locals, outputs, conditionals
- **Best Practices**: Security, organization, CI/CD integration
- **Troubleshooting**: Common issues and resolution strategies
- **Advanced Features**: Workspaces, backends, custom providers

Code Interview Tips:

1. **Start simple** - Begin with basic configuration, then add complexity
2. **Explain your choices** - Why you chose specific approaches
3. **Consider edge cases** - Show awareness of potential issues
4. **Follow best practices** - Demonstrate professional habits

5. **Test your code** - Validate syntax and logic

Common Terraform Interview Patterns:

- Infrastructure provisioning scenarios
- State management challenges
- Module design questions
- CI/CD integration problems
- Troubleshooting exercises
- Best practices discussions

Final Interview Advice:

- **Know the fundamentals** - Understand core Terraform concepts deeply
- **Practice hands-on** - Build real infrastructure with Terraform
- **Stay current** - Know recent Terraform features and updates
- **Think operationally** - Consider maintenance, monitoring, and scaling
- **Emphasize collaboration** - Show how Terraform improves team workflows

Remember: Terraform interviews test both your technical knowledge and your understanding of infrastructure as code principles. Demonstrate not just what Terraform can do, but why and when to use it effectively in real-world scenarios!