Terraform Learning Guide - From Zero to Hero

🎯 What You'll Learn

This guide will take you from complete beginner to confident Terraform user. We'll explain every concept in simple terms, show you real examples, and prepare you for the tricky situations you'll face in real-world infrastructure management.

📚 Chapter 1: Understanding Terraform Basics

What is Terraform?

Imagine you need to set up 100 servers in Azure. You could:

- **Manual Way**: Click through Azure Portal 100 times (boring, error-prone, takes forever)
- **Terraform Way**: Write code once, run it, and watch 100 servers appear automatically

Terraform = Infrastructure as Code (IaC)

Think of it like a recipe book for your infrastructure. You write down what you want, and Terraform makes it happen.

🧱 Chapter 2: Core Concepts Deep Dive

1️⃣ PROVIDER - Your Connection to the Cloud

What is it?
A provider is like a translator between Terraform and your cloud platform (Azure, AWS, GCP). It speaks both languages.

Real-World Analogy:
You're in France but don't speak French. A provider is your translator who helps you order food (create resources).

Code Example:

```
provider "azurerm" {
  features {}
  subscription_id = "your-azure-subscription-id"
}
```

What does this do?

- Connects Terraform to Azure
- Authenticates your identity
- Knows how to create Azure resources

What happens if you forget the provider?

❌ **Error you'll see:**

```
Error: provider "azurerm" is required by resources, but not declared
```

How infrastructure behaves:

- Nothing happens! Terraform refuses to start
- It's like trying to drive without keys in the ignition

How to fix:

1. Add the provider block at the top of your file
2. Run `terraform init` to download the provider plugin
3. Verify with `terraform providers` command

⭐ **Real Situation #1: Wrong Provider Version**

Scenario: You used provider version 3.0, but your team uses 2.0. Code breaks!

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"  # This locks to version 3.x
    }
  }
}
```

Fix: Always pin your provider version to avoid surprises.

---

2 RESOURCE - The Building Blocks

What is it?
Resources are the actual things you want to create: servers, databases, networks, etc.

Real-World Analogy:
If Terraform is a construction company, resources are the actual buildings, roads, and bridges they construct.

Code Example:

```
resource "azurerm_resource_group" "my_group" {
  name     = "my-resources"
  location = "East US"

  tags = {
    Environment = "Learning"
    Owner       = "YourName"
  }
}
```

Breaking it down:

- `azurerm_resource_group` = Type of resource (like "house" or "apartment")
- `"my_group"` = Local name (how you refer to it in code)
- `name = "my-resources"` = Actual name in Azure
- Everything inside `{}` = Properties of the resource

How resources work during provisioning:

1. **Planning Phase:**

   ```
   Terraform Plan:
   + azurerm_resource_group.my_group will be created
     + name     = "my-resources"
     + location = "East US"
   ```

2. **Apply Phase:**

   ```
   azurerm_resource_group.my_group: Creating...
   azurerm_resource_group.my_group: Creation complete after 2s
   ```

3. **In Azure:**
   - A new resource group appears named "my-resources"
   - Located in East US region
   - Tagged with Environment and Owner

What happens if resource creation fails?

❌ **Common Failure Scenario:**

```
Error: A resource with the ID "/subscriptions/.../resourceGroups/my-resources" already
exists
```

How infrastructure behaves:

- Terraform stops immediately
- Already created resources remain (Terraform is atomic per resource)
- State file shows partial completion
- You can't move forward until you fix it

⭐ **Real Situation #2: Resource Already Exists**

Problem: Someone manually created "my-resources" in Azure Portal before you ran Terraform.

Three ways to fix:

Option 1: Import the existing resource

```
# Tell Terraform: "Hey, this resource already exists, track it!"
terraform import azurerm_resource_group.my_group
/subscriptions/SUBSCRIPTION_ID/resourceGroups/my-resources

# Now Terraform manages it
terraform plan  # Should show no changes
```

Option 2: Rename your resource

```
resource "azurerm_resource_group" "my_group" {
  name     = "my-resources-terraform"  # Different name
  location = "East US"
}
```

Option 3: Delete the manual resource

```
# In Azure CLI
az group delete --name my-resources --yes
```

⭐ Real Situation #3: Modifying Existing Resources

Scenario: Your VM is too small. Users complaining it's slow!

Original code:

```
resource "azurerm_linux_virtual_machine" "web_server" {
  name                = "web-vm-01"
  size                = "Standard_B1s"  # Too small!
  # ... other config
}
```

Modified code:

```
resource "azurerm_linux_virtual_machine" "web_server" {
  name                = "web-vm-01"
  size                = "Standard_B2s"  # Bigger!
  # ... other config
}
```

What happens when you run `terraform plan`:

```
Terraform will perform the following actions:

  # azurerm_linux_virtual_machine.web_server must be replaced
-/+ resource "azurerm_linux_virtual_machine" "web_server" {
      ~ size = "Standard_B1s" -> "Standard_B2s"
      # (forces replacement)
    }

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

⚠️ **DANGER: "Forces replacement" means:**

- Old VM will be DELETED
- New VM will be created
- You'll lose all data on the VM unless backed up!
- There will be downtime!

How to handle this safely:

```
resource "azurerm_linux_virtual_machine" "web_server" {
  name                 = "web-vm-01"
  size                 = "Standard_B2s"

  # Prevent accidental deletion
  lifecycle {
    prevent_destroy = true
    create_before_destroy = true  # Create new before deleting old
  }
}
```

3 VARIABLES - Making Code Flexible

What is it?
Variables let you reuse the same code with different values. Like function parameters in programming.

Real-World Analogy:
A coffee machine recipe. Variables are: coffee type, size, sugar amount. Same machine, different outputs.

Code Example:

```
# variables.tf
variable "vm_count" {
  description = "How many VMs to create"
  type        = number
  default     = 3

  validation {
    condition     = var.vm_count >= 1 && var.vm_count <= 10
    error_message = "Must create between 1 and 10 VMs"
  }
}

variable "environment" {
  description = "Environment name"
  type        = string
  # No default = user must provide value
}

variable "vm_sizes" {
  description = "List of VM sizes"
  type        = list(string)
  default     = ["Standard_B1s", "Standard_B2s"]
}

variable "tags" {
  description = "Resource tags"
  type        = map(string)
  default     = {
    ManagedBy = "Terraform"
  }
}
```

Using variables:

```
# main.tf
resource "azurerm_linux_virtual_machine" "web" {
  count = var.vm_count  # Creates 3 VMs
  name  = "vm-${count.index + 1}"
  size  = var.vm_sizes[0]

  tags = merge(var.tags, {
    Environment = var.environment
  })
}
```

Three ways to provide variable values:

Method 1: terraform.tfvars file

```
# terraform.tfvars
vm_count    = 5
environment = "production"
```

Method 2: Command line

```
terraform apply -var="vm_count=5" -var="environment=production"
```

Method 3: Environment variables

```
export TF_VAR_vm_count=5
export TF_VAR_environment=production
terraform apply
```

⭐ **Real Situation #4: Missing Required Variable**

Error:

```
Error: No value for required variable

  on variables.tf line 10:
  10: variable "environment" {

The variable "environment" is required, but no definition was found.
```

How infrastructure behaves:

- Terraform refuses to proceed
- Interactive prompt appears asking for value (annoying!)
- CI/CD pipeline fails

Fixes:

```
# Quick fix: provide on command line
terraform apply -var="environment=dev"

# Better fix: create terraform.tfvars
echo 'environment = "dev"' > terraform.tfvars

# Best fix: use workspace-specific files
# dev.tfvars, staging.tfvars, prod.tfvars
terraform apply -var-file="dev.tfvars"
```

⭐ **Real Situation #5: Variable Type Mismatch**

Problem:

```
variable "vm_count" {
  type = number
}
```

User provides:

```
vm_count = "five"  # String, not number!
```

Error:

```
Error: Invalid value for variable

Expected a number value, got string "five"
```

Fix: Always validate input with validation blocks:

```
variable "vm_count" {
  type    = number
  default = 1

  validation {
    condition     = var.vm_count >= 1 && var.vm_count <= 100
    error_message = "vm_count must be a number between 1 and 100"
  }
}
```

4️⃣ STATE FILE - Terraform's Memory

What is it?
A JSON file where Terraform remembers what it created. Without it, Terraform is like someone with amnesia.

Real-World Analogy:
You're building LEGO. The state file is your instruction booklet with checkmarks showing which steps you completed.

What's inside state file?

```
{
  "resources": [
    {
      "type": "azurerm_resource_group",
      "name": "my_group",
      "instances": [{
        "attributes": {
          "id": "/subscriptions/.../resourceGroups/my-resources",
          "name": "my-resources",
          "location": "eastus"
        }
      }]
    }
  ]
}
```

Why state is CRITICAL:

Scenario without state:

```
Day 1: terraform apply → Creates VM
Day 2: terraform apply → Creates ANOTHER VM (doesn't remember first one!)
Result: Duplicate resources, chaos!
```

Scenario with state:

```
Day 1: terraform apply → Creates VM, records in state
Day 2: terraform apply → Checks state, sees VM exists, does nothing
Result: Infrastructure matches code!
```

⭐ Real Situation #6: Lost State File

Disaster Scenario:

```
Your laptop crashes. State file was stored locally. Now Terraform thinks nothing exists!
```

What happens:

```
terraform plan

# Shows:
Plan: 50 to add, 0 to change, 0 to destroy

# But those 50 resources already exist in Azure!
```

If you run terraform apply:

- Tries to create everything again
- Gets "already exists" errors
- Infrastructure is now orphaned (Terraform can't manage it)

Recovery options:

Option 1: Restore from backup

```
# Azure Blob Storage automatically versions state files
az storage blob download \
  --account-name tfstate \
  --container-name tfstate \
  --name terraform.tfstate \
  --file terraform.tfstate
```

Option 2: Rebuild state with imports

```
# Import each resource one by one (tedious but works)
terraform import azurerm_resource_group.my_group /subscriptions/.../resourceGroups/my-
resources
terraform import azurerm_virtual_network.main /subscriptions/.../virtualNetworks/my-vnet
# ... repeat for all resources
```

Option 3: Start fresh (nuclear option)

```
# Destroy everything manually in Azure
# Remove all .tf files
# Start over (only if resources aren't important)
```

Prevention - Use Remote State:

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

Benefits of remote state:

- Automatic backups and versioning
- Team collaboration (no "which state file is correct?")
- State locking (prevents corruption)
- Encrypted storage

⭐ **Real Situation #7: State File Conflicts**

Scenario: Two team members run terraform apply simultaneously.

Without state locking:

```
Person A: terraform apply → Starts creating resources
Person B: terraform apply → Starts at same time
Result: Both write to state file simultaneously
Outcome: Corrupted state file, infrastructure chaos!
```

With state locking (Azure Blob Storage):

```
Person A: terraform apply → Acquires lock
Person B: terraform apply → Gets error:

Error: Error locking state: state is locked
Lock Info:
  ID:        a1b2c3d4-e5f6-7890-1234-567890abcdef
  Path:      production.tfstate
  Operation: OperationTypeApply
  Who:       person-a@company.com
  Created:   2025-09-30 10:30:00
```

How to handle:

```
# Person B waits for Person A to finish
# Or in emergency, force unlock (dangerous!)
terraform force-unlock a1b2c3d4-e5f6-7890-1234-567890abcdef

# But always communicate first!
```

5 DATA SOURCES - Reading Existing Infrastructure

What is it?

Data sources let Terraform fetch information about things that already exist.

Real-World Analogy:

You're moving into an apartment building. You need to know the address, floor numbers, and mailbox location. Data sources give you that info.

Code Example:

```
# Look up existing resource group
data "azurerm_resource_group" "existing" {
  name = "production-rg"
}

# Use its properties
resource "azurerm_virtual_network" "new" {
  name                = "my-vnet"
  resource_group_name = data.azurerm_resource_group.existing.name
  location            = data.azurerm_resource_group.existing.location
  address_space       = ["10.0.0.0/16"]
}
```

Key difference:

Resources	Data Sources
Create/Modify/Delete	Read only
`resource "type" "name"`	`data "type" "name"`
Terraform manages	Terraform just reads

⭐ Real Situation #8: Data Source Not Found

Code:

```
data "azurerm_resource_group" "existing" {
  name = "production-rg"
}
```

Error:

```
Error: Resource Group "production-rg" was not found
```

How infrastructure behaves:

- Terraform stops completely
- Can't proceed because it can't find the resource
- All dependent resources are blocked

Common causes:

1. Typo in name (`production-rg` vs `production-rgs`)
2. Wrong Azure subscription
3. Resource doesn't exist yet
4. Insufficient permissions to view resource

Fixes:

```
# Verify resource exists
az group show --name production-rg

# Check subscription
az account show

# List all resource groups
az group list --output table

# Fix permissions
az role assignment create \
  --assignee user@example.com \
  --role Reader \
  --scope /subscriptions/SUBSCRIPTION_ID
```

⭐ Real Situation #9: Dynamic Data Sources

Smart use case:

```
# Get latest Ubuntu image automatically
data "azurerm_platform_image" "ubuntu" {
  location  = "East US"
  publisher = "Canonical"
  offer     = "0001-com-ubuntu-server-jammy"
  sku       = "22_04-lts-gen2"
}

# Use in VM (always gets latest!)
resource "azurerm_linux_virtual_machine" "web" {
  name = "web-vm"
  # ... other config

  source_image_reference {
    publisher = data.azurerm_platform_image.ubuntu.publisher
    offer     = data.azurerm_platform_image.ubuntu.offer
    sku       = data.azurerm_platform_image.ubuntu.sku
    version   = "latest"
  }
}
```

Why this matters:

- No hardcoded image IDs

- Always uses latest secure image
- Works across regions automatically

6️⃣ OUTPUTS - Sharing Information

What is it?
Outputs display information after Terraform runs. Like return values from a function.

Real-World Analogy:
After building a house, the contractor gives you the address, keys, and alarm code. Those are outputs.

Code Example:

```
output "vm_public_ip" {
  description = "Public IP to access the web server"
  value       = azurerm_public_ip.web.ip_address
}

output "database_connection_string" {
  description = "Connection string for the database"
  value       = azurerm_sql_server.main.fully_qualified_domain_name
  sensitive   = true  # Won't show in logs
}

output "all_vm_names" {
  description = "Names of all VMs created"
  value       = azurerm_linux_virtual_machine.web[*].name
}
```

After terraform apply:

```
Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

vm_public_ip = "20.123.45.67"
all_vm_names = [
  "web-vm-1",
  "web-vm-2",
  "web-vm-3"
]
database_connection_string = <sensitive>
```

Viewing outputs later:

```
# Show all outputs
terraform output

# Show specific output
terraform output vm_public_ip

# Show sensitive output
terraform output -raw database_connection_string

# Output as JSON (for scripts)
terraform output -json > infrastructure.json
```

⭐ **Real Situation #10: Outputs Referencing Destroyed Resources**

Problem:

```
output "vm_ip" {
  value = azurerm_public_ip.web.ip_address
}
```

You delete the public IP:

```
# Commented out or removed
# resource "azurerm_public_ip" "web" { ... }
```

Error:

```
Error: Reference to undeclared resource

A managed resource "azurerm_public_ip" "web" has not been declared
```

Fix:

```
# Make output conditional
output "vm_ip" {
  value = try(azurerm_public_ip.web.ip_address, "No public IP")
}

# Or remove the output entirely
```

🏗 Chapter 3: Real-World Azure VM Provisioning

Provisioning 5 Azure VMs - Step by Step

Business scenario: You need 5 web servers behind a load balancer for your e-commerce site.

Step 1: Understanding the architecture

```
Internet
   ↓
Load Balancer (Public IP)
   ↓
Backend Pool
   ↓
├─ VM-1 ─┐
├─ VM-2 ─┤
├─ VM-3 ─┼─ Virtual Network
├─ VM-4 ─┤   Subnet: 10.0.1.0/24
└─ VM-5 ─┘
```

Step 2: Building the foundation

```
# Step 2a: Create resource group (container for everything)
resource "azurerm_resource_group" "web" {
  name     = "rg-web-production"
  location = "East US"

  tags = {
    Environment = "Production"
    Project     = "ECommerce"
    CostCenter  = "IT"
  }
}

# What happens when you apply this:
# 1. Terraform contacts Azure API
# 2. Azure creates empty resource group
# 3. State file records: "rg-web-production exists in East US"
# 4. Takes ~2 seconds
```

Step 3: Network setup

```
# Step 3a: Virtual Network (like a private network in Azure)
resource "azurerm_virtual_network" "main" {
  name                = "vnet-web"
  address_space       = ["10.0.0.0/16"]  # 65,536 IP addresses available
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name

  tags = azurerm_resource_group.web.tags
}

# Step 3b: Subnet (section of the network for web servers)
resource "azurerm_subnet" "web" {
  name                 = "subnet-web"
  resource_group_name  = azurerm_resource_group.web.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.1.0/24"]  # 256 IP addresses for VMs
}
```

```
# What happens during provisioning:
# 1. VNet created first (Terraform detects dependency)
# 2. Azure allocates IP range 10.0.0.0/16
# 3. Subnet carved out from VNet
# 4. Takes ~10-15 seconds
# 5. State file records both resources
```

Step 4: Security (Firewall rules)

```
resource "azurerm_network_security_group" "web" {
  name                = "nsg-web"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name

  # Allow HTTP traffic from internet
  security_rule {
    name                       = "AllowHTTP"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "80"
    source_address_prefix      = "*"  # From anywhere
    destination_address_prefix = "*"  # To any VM
  }

  # Allow HTTPS
  security_rule {
    name                       = "AllowHTTPS"
    priority                   = 110
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "443"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }

  # Block everything else (implicit deny all)
}

# Associate NSG with subnet
resource "azurerm_subnet_network_security_group_association" "web" {
  subnet_id                 = azurerm_subnet.web.id
  network_security_group_id = azurerm_network_security_group.web.id
}
```

⭐ Real Situation #11: Forgot to Associate NSG

What happens:

```
You create NSG with rules
But forget the association
Result: VMs have NO firewall protection!
Anyone can access any port!
```

Detection:

```
# In Azure Portal, subnet shows "No NSG"
# Or test: Can SSH to VM on port 22 (shouldn't be allowed)
```

Fix:

```
# Run terraform plan
# Notice association resource missing
# Add association resource and apply
terraform apply
```

Step 5: Load Balancer

```
# Public IP for load balancer
resource "azurerm_public_ip" "lb" {
  name                = "pip-lb-web"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name
  allocation_method   = "Static"
  sku                 = "Standard"
}

# Load balancer
resource "azurerm_lb" "web" {
  name                = "lb-web"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name
  sku                 = "Standard"

  frontend_ip_configuration {
    name                 = "PublicIPAddress"
    public_ip_address_id = azurerm_public_ip.lb.id
  }
}

# Backend pool (where VMs will be added)
resource "azurerm_lb_backend_address_pool" "web" {
  loadbalancer_id = azurerm_lb.web.id
  name            = "BackendPool"
}

# Health probe (checks if VMs are healthy)
resource "azurerm_lb_probe" "web" {
  loadbalancer_id = azurerm_lb.web.id
  name            = "http-probe"
  protocol        = "Http"
  request_path    = "/health"  # App must respond to this
  port            = 80
}

# Load balancer rule
resource "azurerm_lb_rule" "web" {
  loadbalancer_id                = azurerm_lb.web.id
  name                           = "HTTPRule"
  protocol                       = "Tcp"
  frontend_port                  = 80
  backend_port                   = 80
  frontend_ip_configuration_name = "PublicIPAddress"
  backend_address_pool_ids       = [azurerm_lb_backend_address_pool.web.id]
  probe_id                       = azurerm_lb_probe.web.id
}
```

How load balancing works:

```
User request → Load Balancer Public IP (52.123.45.67)
                ↓
          Health check probe
                ↓
    Finds healthy VMs (VMs responding to /health)
                ↓
    Distributes traffic evenly:
    - Request 1 → VM-1
    - Request 2 → VM-2
    - Request 3 → VM-3
    - Request 4 → VM-4
    - Request 5 → VM-5
    - Request 6 → VM-1 (round-robin)
```

⭐ Real Situation #12: Health Probe Failing

Symptoms:

```
Load balancer created successfully
VMs running
But users can't access website!
```

Diagnosis:

```
# Check health probe status in Azure
az network lb show \
  --resource-group rg-web-production \
  --name lb-web \
  --query "probes[].{Name:name, Port:port, Path:requestPath}"

# Result shows: All VMs unhealthy
```

Root causes:

1. VMs don't have `/health` endpoint
2. Web server not started
3. Firewall blocking health probe
4. Wrong port configured

Fix:

```
# Option 1: Change probe to simpler check
resource "azurerm_lb_probe" "web" {
  loadbalancer_id = azurerm_lb.web.id
  name            = "tcp-probe"
  protocol        = "Tcp"  # Just check if port open
  port            = 80
  # No request_path needed
}

# Option 2: Ensure VMs have health endpoint
resource "azurerm_linux_virtual_machine" "web" {
  # ... other config

  custom_data = base64encode(<<-EOF
    #!/bin/bash
    apt-get update
    apt-get install -y nginx

    # Create health endpoint
    echo "OK" > /var/www/html/health

    systemctl start nginx
  EOF
  )
}
```

Step 6: Creating 5 VMs

```
# Network interfaces (one per VM)
resource "azurerm_network_interface" "web" {
  count               = 5  # Creates 5 NICs
  name                = "nic-web-${count.index + 1}"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.web.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

```
# Connect NICs to load balancer
resource "azurerm_network_interface_backend_address_pool_association" "web" {
  count                   = 5
  network_interface_id    = azurerm_network_interface.web[count.index].id
  ip_configuration_name   = "internal"
  backend_address_pool_id = azurerm_lb_backend_address_pool.web.id
}

# The 5 VMs!
resource "azurerm_linux_virtual_machine" "web" {
  count               = 5
  name                = "vm-web-${count.index + 1}"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name
  size                = "Standard_B2s"
  admin_username      = "azureuser"

  # SSH key authentication (more secure than password)
  admin_ssh_key {
    username   = "azureuser"
    public_key = file("~/.ssh/id_rsa.pub")
  }

  network_interface_ids = [
    azurerm_network_interface.web[count.index].id,
  ]

  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Premium_LRS"
    disk_size_gb         = 30
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "0001-com-ubuntu-server-jammy"
    sku       = "22_04-lts-gen2"
    version   = "latest"
  }

  # Install nginx on each VM
  custom_data = base64encode(<<-EOF
    #!/bin/bash
    apt-get update
    apt-get install -y nginx

    # Create unique page per VM
    echo "<h1>Web Server ${count.index + 1}</h1>" > /var/www/html/index.html
    echo "OK" > /var/www/html/health

    systemctl start nginx
    systemctl enable nginx
  EOF
  )
}

# Output the load balancer IP
output "load_balancer_ip" {
  value = azurerm_public_ip.lb.ip_address
}
```

What happens during provisioning:

```
terraform apply

Timeline of events:
[0:00] Planning phase begins
[0:05] Plan complete: 20 resources to add
[0:06] User confirms: yes
[0:07] Resource group created ✓
[0:10] Virtual network created ✓
[0:15] Subnet created ✓
[0:20] NSG created ✓
[0:25] Public IP allocated ✓
[0:30] Load balancer created ✓
[0:35] Backend pool created ✓
[0:40] Health probe configured ✓
[0:45] NIC-1 created ✓
[0:46] NIC-2 created ✓
[0:47] NIC-3 created ✓
[0:48] NIC-4 created ✓
[0:49] NIC-5 created ✓
[1:00] VM-1 creating (parallel) 🔄
[1:00] VM-2 creating (parallel) 🔄
[1:00] VM-3 creating (parallel) 🔄
[1:00] VM-4 creating (parallel) 🔄
[1:00] VM-5 creating (parallel) 🔄
[3:00] VM-1 complete ✓ (running custom_data script)
[3:10] VM-2 complete ✓
[3:15] VM-3 complete ✓
[3:20] VM-4 complete ✓
[3:25] VM-5 complete ✓
[3:30] NICs associated with backend pool ✓
[3:35] Apply complete!

Outputs:
load_balancer_ip = "52.123.45.67"
```

Testing your deployment:

```
# Wait 2 minutes for VMs to finish setup
sleep 120

# Test load balancer
curl http://52.123.45.67

# Should see rotating responses:
# <h1>Web Server 1</h1>
# <h1>Web Server 2</h1>
# <h1>Web Server 3</h1>
# etc.
```

⭐ Real Situation #13: VMs Created But Not Responding

Problem:

```
All resources created successfully
Load balancer shows healthy
But curl http://52.123.45.67 times out!
```

Debugging steps:

Step 1: Check NSG rules

```
az network nsg show \
  --resource-group rg-web-production \
  --name nsg-web \
  --query "securityRules[].{Name:name, Port:destinationPortRange, Access:access}"

# Verify port 80 is allowed
```

Step 2: Check if VMs are actually running

```
az vm list \
  --resource-group rg-web-production \
  --show-details \
  --query "[].{Name:name, PowerState:powerState}"

# All should show "VM running"
```

Step 3: SSH into a VM to check nginx

```
# First, add SSH rule temporarily to NSG
az network nsg rule create \
  --resource-group rg-web-production \
  --nsg-name nsg-web \
  --name AllowSSH \
  --priority 1000 \
  --destination-port-range 22 \
  --access Allow
```

```
# Get VM private IP
az vm show \
  --resource-group rg-web-production \
  --name vm-web-1 \
  --show-details \
  --query privateIps

# SSH to VM (from within Azure)
ssh azureuser@10.0.1.4

# Inside VM: Check nginx
sudo systemctl status nginx
curl localhost

# If not running:
sudo systemctl start nginx
```

Common root causes:

1. `custom_data` script failed (typo in bash script)
2. Nginx installation failed (network issue during apt-get)
3. VMs need more time to initialize
4. Wrong subnet - VMs not in same subnet as health probe can reach

⭐ Real Situation #14: One VM Keeps Failing

Scenario:

```
VM-1: ✓ Created
VM-2: ✓ Created
VM-3: ❌ Failed
VM-4: ⏸ Waiting (blocked)
VM-5: ⏸ Waiting (blocked)

Error: compute.VirtualMachinesClient#CreateOrUpdate:
Failure sending request: StatusCode=409
Code="OperationNotAllowed"
Message="The selected VM size 'Standard_B2s' is not available in zone '3'."
```

What this means:

- Azure ran out of B2s VMs in availability zone 3
- VM-3 can't be created
- Terraform stops (all or nothing per resource type)
- VM-4 and VM-5 never get created

Fix Option 1: Remove availability zones

```
resource "azurerm_linux_virtual_machine" "web" {
  count = 5
  name  = "vm-web-${count.index + 1}"
  # ... other config

  # Remove this line:
  # zone = (count.index % 3) + 1

  # Or use availability set instead
  availability_set_id = azurerm_availability_set.web.id
}

resource "azurerm_availability_set" "web" {
  name                = "avset-web"
  location            = azurerm_resource_group.web.location
  resource_group_name = azurerm_resource_group.web.name
}
```

Fix Option 2: Use different VM size

```
resource "azurerm_linux_virtual_machine" "web" {
  count = 5
  name  = "vm-web-${count.index + 1}"
  size  = "Standard_D2s_v3"  # More widely available
  # ... rest of config
}
```

Fix Option 3: Deploy to different region

```
variable "location" {
  default = "West US 2"  # Try different region
}
```

⭐ Real Situation #15: Need to Add More VMs Later

Original deployment: 5 VMs
Business needs: Scale to 10 VMs

Naive approach (WRONG):

```
# Change count from 5 to 10
resource "azurerm_linux_virtual_machine" "web" {
  count = 10  # Changed from 5
  # ... rest of config
}
```

What happens:

```
terraform plan

# Shows:
  # azurerm_linux_virtual_machine.web[5] will be created
  # azurerm_linux_virtual_machine.web[6] will be created
  # azurerm_linux_virtual_machine.web[7] will be created
  # azurerm_linux_virtual_machine.web[8] will be created
  # azurerm_linux_virtual_machine.web[9] will be created

Plan: 5 to add, 0 to change, 0 to destroy
```

This is correct! Terraform intelligently:

1. Sees existing VMs (web[0] through web[4])
2. Only creates new ones (web[5] through web[9])
3. Leaves existing VMs untouched

Smart approach (BETTER):

```
# Use variable for easy changes
variable "vm_count" {
  description = "Number of web servers"
  type        = number
  default     = 5
}

resource "azurerm_linux_virtual_machine" "web" {
  count = var.vm_count
  # ... rest of config
}

# Scale up
# terraform apply -var="vm_count=10"

# Scale down (dangerous!)
# terraform apply -var="vm_count=3"
# This DELETES vm-web-4 and vm-web-5!
```

🌐 Chapter 4: Azure Networking Deep Dive

Understanding Azure Networking Components

The networking hierarchy:

```
Azure Subscription
  └── Resource Group
      └── Virtual Network (VNet)
          ├── Subnet 1 (Web Tier)
          ├── Subnet 2 (App Tier)
          └── Subnet 3 (Data Tier)
              ├── Network Interface 1
              ├── Network Interface 2
              └── Network Interface 3
                  └── Virtual Machines
```

Scenario: Building a Secure 3-Tier Application

Architecture:

```
Internet → Load Balancer → Web Tier (Public Subnet)
                              ↓
                           App Tier (Private Subnet)
                              ↓
                           Database Tier (Private Subnet)
```

Complete networking configuration:

```
# 1. Virtual Network (The Big Container)
resource "azurerm_virtual_network" "main" {
  name                = "vnet-app"
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name
  address_space       = ["10.0.0.0/16"]  # 65,536 IPs

  # DNS servers (optional)
  dns_servers = ["10.0.0.4", "10.0.0.5"]

  tags = {
    Environment = "Production"
  }
}

# What this creates:
# - Isolated network in Azure
# - IP range: 10.0.0.1 to 10.0.255.254
# - No internet access by default (you add that)
```

Understanding address spaces:

```
10.0.0.0/16 = 65,536 IP addresses
  ├── 10.0.0.0/24 = 256 IPs (Subnet for Web tier)
  ├── 10.0.1.0/24 = 256 IPs (Subnet for App tier)
  ├── 10.0.2.0/24 = 256 IPs (Subnet for Database)
  └── 10.0.3.0/24 = 256 IPs (Subnet for Management)
```

2. Subnets (Segmenting the network)

```
# Web tier subnet (public-facing)
resource "azurerm_subnet" "web" {
  name                 = "subnet-web"
  resource_group_name  = azurerm_resource_group.network.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.0.0/24"]

  # Service endpoints (direct connection to Azure services)
  service_endpoints = ["Microsoft.Storage", "Microsoft.KeyVault"]
}

# App tier subnet (private)
resource "azurerm_subnet" "app" {
  name                 = "subnet-app"
  resource_group_name  = azurerm_resource_group.network.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.1.0/24"]

  # No public IPs allowed in this subnet
  private_endpoint_network_policies_enabled = true
}

# Database tier subnet (most private)
resource "azurerm_subnet" "data" {
  name                 = "subnet-data"
  resource_group_name  = azurerm_resource_group.network.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.2.0/24"]

  # Delegate to Azure SQL
  delegation {
    name = "sql-delegation"
    service_delegation {
      name = "Microsoft.Sql/managedInstances"
      actions = [
        "Microsoft.Network/virtualNetworks/subnets/join/action",
      ]
    }
  }
}
```

⭐ Real Situation #16: Overlapping Subnets

Mistake:

```
resource "azurerm_subnet" "web" {
  address_prefixes = ["10.0.0.0/24"]  # 10.0.0.1 - 10.0.0.254
}

resource "azurerm_subnet" "app" {
  address_prefixes = ["10.0.0.0/25"]  # 10.0.0.1 - 10.0.0.126 OVERLAP!
}
```

Error:

```
Error: Subnet address range overlaps with existing subnet
```

Why this matters:

- IP addresses would conflict
- Routing becomes impossible
- Azure rejects the configuration

Fix:

```
resource "azurerm_subnet" "web" {
  address_prefixes = ["10.0.0.0/24"]  # 10.0.0.x
}

resource "azurerm_subnet" "app" {
  address_prefixes = ["10.0.1.0/24"]  # 10.0.1.x (different)
}
```

3. Network Security Groups (Firewalls)

```
# NSG for web tier
resource "azurerm_network_security_group" "web" {
  name                = "nsg-web"
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name

  # Rule 1: Allow HTTP from internet
  security_rule {
    name                       = "AllowHTTP"
    priority                   = 100  # Lower = higher priority
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "80"
    source_address_prefix      = "*"  # From anywhere
    destination_address_prefix = "*"
  }

  # Rule 2: Allow HTTPS from internet
  security_rule {
    name                       = "AllowHTTPS"
    priority                   = 110
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "443"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }

  # Rule 3: Allow SSH from corporate network only
  security_rule {
    name                       = "AllowSSH"
    priority                   = 120
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "22"
    source_address_prefix      = "203.0.113.0/24"  # Your office IP
    destination_address_prefix = "*"
  }

  # Implicit deny all (Azure adds this automatically)
  # Any traffic not matching above rules is blocked
}

# NSG for app tier
resource "azurerm_network_security_group" "app" {
  name                = "nsg-app"
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name

  # Only allow traffic from web tier
  security_rule {
    name                       = "AllowFromWeb"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "8080"
    source_address_prefix      = "10.0.0.0/24"  # Web subnet only
    destination_address_prefix = "*"
  }

  # Block everything else
  security_rule {
    name                       = "DenyAll"
    priority                   = 4096  # Lowest priority
    direction                  = "Inbound"
    access                     = "Deny"
    protocol                   = "*"
    source_port_range          = "*"
    destination_port_range     = "*"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }
}

# NSG for database tier
resource "azurerm_network_security_group" "data" {
  name                = "nsg-data"
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name

  # Only allow database traffic from app tier
  security_rule {
    name                       = "AllowSQLFromApp"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "1433"  # SQL Server port
    source_address_prefix      = "10.0.1.0/24"  # App subnet only
    destination_address_prefix = "*"
  }
}

# Associate NSGs with subnets
resource "azurerm_subnet_network_security_group_association" "web" {
  subnet_id                 = azurerm_subnet.web.id
  network_security_group_id = azurerm_network_security_group.web.id
}

resource "azurerm_subnet_network_security_group_association" "app" {
  subnet_id                 = azurerm_subnet.app.id
  network_security_group_id = azurerm_network_security_group.app.id
}

resource "azurerm_subnet_network_security_group_association" "data" {
  subnet_id                 = azurerm_subnet.data.id
  network_security_group_id = azurerm_network_security_group.data.id
}
```

How NSG rules are evaluated:

```
Incoming request to VM on port 80:
1. Check Priority 100 rule: HTTP allowed? → YES → Allow traffic ✓
   (Stop checking, request allowed)

Incoming request to VM on port 22 from 203.0.113.50:
1. Check Priority 100: HTTP? → NO
2. Check Priority 110: HTTPS? → NO
```

```
3. Check Priority 120: SSH from 203.0.113.0/24? → YES → Allow ✓

Incoming request to VM on port 22 from 1.2.3.4:
1. Check Priority 100: HTTP? → NO
2. Check Priority 110: HTTPS? → NO
3. Check Priority 120: SSH from 203.0.113.0/24? → NO (wrong source)
4. Implicit deny all → Block ✗
```

⭐ **Real Situation #17: Can't SSH to VM After Applying NSG**

Symptoms:

```
ssh azureuser@vm-web-1
# Connection times out
```

Debugging:

Step 1: Check if NSG exists and is associated

```
az network nsg list --resource-group rg-web-production

# Check association
az network vnet subnet show \
  --resource-group rg-web-production \
  --vnet-name vnet-app \
  --name subnet-web \
  --query networkSecurityGroup
```

Step 2: Check NSG rules

```
az network nsg rule list \
  --resource-group rg-web-production \
  --nsg-name nsg-web \
  --output table

# Look for SSH rule (port 22)
```

Step 3: Test from correct source IP

```
# Check your public IP
curl ifconfig.me
# Returns: 203.0.113.55

# If NSG only allows 203.0.113.0/24, and you're 203.0.113.55:
# ✓ Should work

# If you're 198.51.100.10:
# ✗ Won't work (need to add rule)
```

Fix:

```
# Add your current IP to NSG
resource "azurerm_network_security_group" "web" {
  # ... existing rules

  security_rule {
    name                       = "AllowSSHFromMyIP"
    priority                   = 130
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "22"
    source_address_prefix      = "198.51.100.10/32"  # Your IP
    destination_address_prefix = "*"
  }
}

# Apply change
terraform apply

# Test again
ssh azureuser@vm-web-1  # Should work now
```

4. Route Tables (Traffic Direction)

```
# Route table for app tier
resource "azurerm_route_table" "app" {
  name                = "rt-app"
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name

  # Force all internet traffic through firewall
  route {
    name                   = "InternetViaFirewall"
    address_prefix         = "0.0.0.0/0"  # All internet traffic
    next_hop_type          = "VirtualAppliance"
    next_hop_in_ip_address = "10.0.3.4"  # Firewall IP
  }

  # Local traffic stays local
  route {
    name           = "LocalTraffic"
    address_prefix = "10.0.0.0/16"  # Entire VNet
    next_hop_type  = "VnetLocal"
  }
}

# Associate route table
resource "azurerm_subnet_route_table_association" "app" {
  subnet_id      = azurerm_subnet.app.id
  route_table_id = azurerm_route_table.app.id
}
```

How routing works:

```
App tier VM wants to access google.com:
1. VM sends packet to 172.217.164.46 (Google IP)
2. Checks route table:
   - Does it match 10.0.0.0/16? NO
   - Does it match 0.0.0.0/0? YES → Send to firewall at 10.0.3.4
3. Firewall inspects traffic, allows, forwards to internet
4. Response comes back through firewall to VM
```

5. VNet Peering (Connecting Networks)

```
# Hub VNet (central network)
resource "azurerm_virtual_network" "hub" {
  name                = "vnet-hub"
  address_space       = ["10.0.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name
}

# Spoke VNet 1 (production)
resource "azurerm_virtual_network" "spoke1" {
  name                = "vnet-prod"
  address_space       = ["10.1.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name
}

# Spoke VNet 2 (development)
resource "azurerm_virtual_network" "spoke2" {
  name                = "vnet-dev"
  address_space       = ["10.2.0.0/16"]
  location            = "East US"
  resource_group_name = azurerm_resource_group.network.name
}

# Peer hub to spoke1
resource "azurerm_virtual_network_peering" "hub_to_spoke1" {
  name                      = "hub-to-prod"
  resource_group_name       = azurerm_resource_group.network.name
  virtual_network_name      = azurerm_virtual_network.hub.name
  remote_virtual_network_id = azurerm_virtual_network.spoke1.id

  allow_virtual_network_access = true
  allow_forwarded_traffic      = true
  allow_gateway_transit        = true  # Hub can share VPN gateway
}

# Peer spoke1 to hub (bidirectional)
resource "azurerm_virtual_network_peering" "spoke1_to_hub" {
  name                      = "prod-to-hub"
  resource_group_name       = azurerm_resource_group.network.name
  virtual_network_name      = azurerm_virtual_network.spoke1.name
  remote_virtual_network_id = azurerm_virtual_network.hub.id

  allow_virtual_network_access = true
  allow_forwarded_traffic      = true
  use_remote_gateways          = true  # Use hub's VPN gateway
}
```

Peering topology:

```
         Hub VNet (10.0.0.0/16)
           /          \
          /            \
         /              \
Spoke1 (10.1.0.0/16)  Spoke2 (10.2.0.0/16)
   (Production)          (Development)

- Hub can talk to Spoke1 ✓
- Hub can talk to Spoke2 ✓
- Spoke1 CANNOT talk to Spoke2 ✗ (not peered)
```

⭐ Real Situation #18: Peered VNets Can't Communicate

Setup:

```
Hub VNet: 10.0.0.0/16
Spoke VNet: 10.1.0.0/16
Peering created successfully
```

Problem:

```
# From VM in Spoke VNet
ping 10.0.0.4  # VM in Hub VNet
# Request timeout
```

Common causes:

Cause 1: Forgot bidirectional peering

```
# Only created hub_to_spoke, forgot spoke_to_hub!
# Need BOTH directions
```

Cause 2: NSG blocking traffic

```
# Check NSG rules
az network nsg rule list --nsg-name nsg-hub --output table

# Need to allow traffic from 10.1.0.0/16
```

Cause 3: Peering not in "Connected" state

```
az network vnet peering show \
  --resource-group rg-network \
  --vnet-name vnet-hub \
  --name hub-to-spoke \
  --query peeringState

# Should return: "Connected"
# If "Initiated" or "Disconnected", peering broken
```

Fix:

```
# Ensure both peerings exist and are configured
resource "azurerm_virtual_network_peering" "hub_to_spoke" {
  name                         = "hub-to-spoke"
  resource_group_name          = azurerm_resource_group.network.name
  virtual_network_name         = azurerm_virtual_network.hub.name
  remote_virtual_network_id    = azurerm_virtual_network.spoke.id
  allow_virtual_network_access = true  # CRITICAL: Must be true
}

resource "azurerm_virtual_network_peering" "spoke_to_hub" {
  name                         = "spoke-to-hub"
  resource_group_name          = azurerm_resource_group.network.name
  virtual_network_name         = azurerm_virtual_network.spoke.name
  remote_virtual_network_id    = azurerm_virtual_network.hub.id
  allow_virtual_network_access = true  # CRITICAL: Must be true
}

# Update NSG to allow traffic
resource "azurerm_network_security_group" "hub" {
  # ... existing rules

  security_rule {
    name                       = "AllowFromSpoke"
    priority                   = 150
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "*"
    source_port_range          = "*"
    destination_port_range     = "*"
    source_address_prefix      = "10.1.0.0/16"  # Spoke VNet
    destination_address_prefix = "*"
  }
}
```

💾 Chapter 5: Azure Storage Configuration

Understanding Azure Storage Types

Storage types:

```
Storage Account
  ├── Blob Storage (unstructured data: images, videos, backups)
  ├── File Storage (SMB file shares: shared folders)
  ├── Queue Storage (message queues: async processing)
  ├── Table Storage (NoSQL: key-value data)
  └── Disk Storage (VM disks: managed disks)
```

Real-World Scenario: E-Commerce Storage Architecture

Requirements:

- Product images (Blob)
- Customer uploads (Blob)
- Shared configuration files (File Share)
- Order processing queue (Queue)
- Session data (Table)
- VM disks (Managed Disks)

Complete configuration:

```
# 1. Storage Account
resource "azurerm_storage_account" "ecommerce" {
  name                     = "stecommerce${random_string.suffix.result}"
  resource_group_name      = azurerm_resource_group.storage.name
  location                 = "East US"
  account_tier             = "Standard"  # or "Premium"
  account_replication_type = "GRS"       # Geo-redundant
  account_kind             = "StorageV2"

  # Security
  enable_https_traffic_only      = true
  min_tls_version               = "TLS1_2"
  allow_nested_items_to_be_public = false  # No public containers

  # Network access
  network_rules {
    default_action             = "Deny"  # Block by default
    ip_rules                   = ["203.0.113.0/24"]  # Allow office
    virtual_network_subnet_ids = [azurerm_subnet.web.id]  # Allow web tier
    bypass                     = ["AzureServices"]  # Allow Azure services
  }

  # Blob properties
  blob_properties {
    versioning_enabled  = true  # Keep old versions
    change_feed_enabled = true  # Track changes

    # Soft delete for blobs
    delete_retention_policy {
      days = 30  # Recover deleted blobs within 30 days
    }

    # Soft delete for containers
    container_delete_retention_policy {
      days = 30
    }
  }

  tags = {
    Environment = "Production"
    Purpose     = "E-Commerce"
  }
}

# Generate unique suffix
resource "random_string" "suffix" {
  length  = 8
  special = false
  upper   = false
}
```

What happens during storage account creation:

```
terraform apply

[0:00] Validating name "stecommerce12ab34cd"
[0:02] Checking name availability globally (must be unique)
[0:05] Creating storage account in East US
[0:10] Configuring GRS replication
       ├─ Primary: East US
       └─ Secondary: West US (automatic)
[0:15] Enabling blob versioning
[0:18] Configuring network rules
[0:20] Setting up soft delete policies
[0:25] Storage account ready ✓

Outputs:
primary_blob_endpoint = "https://stecommerce12ab34cd.blob.core.windows.net/"
primary_connection_string = "<sensitive>"
```

2. Blob Containers

```
# Container for product images
resource "azurerm_storage_container" "product_images" {
  name                  = "product-images"
  storage_account_name  = azurerm_storage_account.ecommerce.name
  container_access_type = "private"  # No anonymous access

  metadata = {
    purpose = "Product photos"
  }
}

# Container for customer uploads
resource "azurerm_storage_container" "customer_uploads" {
  name                  = "customer-uploads"
  storage_account_name  = azurerm_storage_account.ecommerce.name
  container_access_type = "private"
}

# Container for backups
resource "azurerm_storage_container" "backups" {
  name                  = "database-backups"
  storage_account_name  = azurerm_storage_account.ecommerce.name
  container_access_type = "private"
}
```

How applications use blob storage:

```python
# Python application example
from azure.storage.blob import BlobServiceClient

# Connection string from Terraform output
connection_string = "<from terraform output storage_connection_string>"
blob_service = BlobServiceClient.from_connection_string(connection_string)

# Upload product image
blob_client = blob_service.get_blob_client(
    container="product-images",
    blob="laptop-001.jpg"
)
with open("laptop-001.jpg", "rb") as data:
    blob_client.upload_blob(data)

# Result: Image stored at:
# https://stecommerce12ab34cd.blob.core.windows.net/product-images/laptop-001.jpg
```

⭐ Real Situation #19: Storage Account Name Already Taken

Error:

```
Error: creating Storage Account "stecommerceprod":
storage.AccountsClient#Create:
Failure sending request: StatusCode=409
Code="StorageAccountAlreadyExists"
Message="The storage account named stecommerceprod is already taken."
```

Why this happens:

- Storage account names are GLOBALLY unique across ALL Azure
- Someone else (or you in another subscription) used this name
- Names are 3-24 characters, lowercase letters and numbers only

Fix with random suffix:

```
resource "random_string" "suffix" {
  length  = 8
  special = false
  upper   = false

  # Regenerate if needed
  keepers = {
    resource_group = azurerm_resource_group.storage.name
  }
}

resource "azurerm_storage_account" "ecommerce" {
  # Guaranteed unique!
  name = "stecommerce${random_string.suffix.result}"
  # stecommerce12ab34cd (unlikely to collide)

  # ... rest of config
}

# Output the actual name
output "storage_account_name" {
  value = azurerm_storage_account.ecommerce.name
}
```

⭐ Real Situation #20: Can't Access Storage from Application

Scenario:

```
Storage account created ✓
Container created ✓
Application trying to upload → Access Denied!
```

Error in application logs:

```
AuthorizationPermissionMismatch: This request is not authorized
Status: 403 (Forbidden)
ErrorCode: AuthorizationPermissionMismatch
```

Debugging steps:

Step 1: Check network rules

```
az storage account show \
  --name stecommerce12ab34cd \
  --resource-group rg-storage \
  --query networkRuleSet

# Returns:
{
  "defaultAction": "Deny",
  "ipRules": [
    {"value": "203.0.113.0/24"}
  ]
}

# Problem: Application running from 198.51.100.50 (not allowed!)
```

Step 2: Fix network access

Option 1: Add application's IP

```
resource "azurerm_storage_account" "ecommerce" {
  # ... other config

  network_rules {
    default_action = "Deny"
    ip_rules = [
      "203.0.113.0/24",    # Office
      "198.51.100.50/32"   # Application server
    ]
    bypass = ["AzureServices"]
  }
}
```

Option 2: Use service endpoints

```
# If application is in Azure VNet
resource "azurerm_storage_account" "ecommerce" {
  # ... other config

  network_rules {
    default_action             = "Deny"
    virtual_network_subnet_ids = [
      azurerm_subnet.web.id,  # Web tier can access
      azurerm_subnet.app.id   # App tier can access
    ]
  }
}

# Add service endpoint to subnet
resource "azurerm_subnet" "app" {
  name                 = "subnet-app"
  # ... other config
  service_endpoints    = ["Microsoft.Storage"]
}
```

Option 3: Use private endpoint (most secure)

```
# Create private endpoint
resource "azurerm_private_endpoint" "storage" {
  name                = "pe-storage"
  location            = azurerm_resource_group.storage.location
  resource_group_name = azurerm_resource_group.storage.name
  subnet_id           = azurerm_subnet.app.id

  private_service_connection {
    name                           = "pe-connection"
    private_connection_resource_id = azurerm_storage_account.ecommerce.id
    subresource_names              = ["blob"]
    is_manual_connection           = false
  }
}

# Now storage is accessible via private IP 10.0.1.X
# No public internet access needed!
```

3. File Shares (Shared Folders)

```
# File share for application configuration
resource "azurerm_storage_share" "config" {
  name                 = "app-config"
  storage_account_name = azurerm_storage_account.ecommerce.name
  quota                = 100   # 100 GB

  metadata = {
    purpose = "Application configuration files"
  }
}

# File share for shared media
resource "azurerm_storage_share" "media" {
  name                 = "shared-media"
  storage_account_name = azurerm_storage_account.ecommerce.name
  quota                = 500   # 500 GB

  # Access tier (for large files)
  access_tier = "Hot"   # or "Cool", "TransactionOptimized"
}

# Directories within file share
resource "azurerm_storage_share_directory" "logs" {
  name                 = "logs"
  share_name           = azurerm_storage_share.config.name
  storage_account_name = azurerm_storage_account.ecommerce.name
}
```

Mounting file share on VM:

```
# On Linux VM
sudo apt-get install cifs-utils
```

```
# Create mount point
sudo mkdir /mnt/appconfig

# Mount file share
sudo mount -t cifs \
  //stecommerce12ab34cd.file.core.windows.net/app-config \
  /mnt/appconfig \
  -o vers=3.0,username=stecommerce12ab34cd,password=<storage-
key>,dir_mode=0777,file_mode=0777

# Make permanent (add to /etc/fstab)
echo "//stecommerce12ab34cd.file.core.windows.net/app-config /mnt/appconfig cifs
vers=3.0,username=stecommerce12ab34cd,password=<key>,dir_mode=0777,file_mode=0777 0 0" |
sudo tee -a /etc/fstab

# Now accessible as normal folder
ls /mnt/appconfig
echo "database_host=10.0.2.5" > /mnt/appconfig/database.conf
```

4. Storage Lifecycle Management

```
# Automatically move data to cheaper storage tiers
resource "azurerm_storage_management_policy" "lifecycle" {
  storage_account_id = azurerm_storage_account.ecommerce.id

  # Rule 1: Archive old product images
  rule {
    name    = "archive-old-images"
    enabled = true

    filters {
      prefix_match = ["product-images/"]
      blob_types   = ["blockBlob"]
    }

    actions {
      base_blob {
        # Move to cool storage after 30 days
        tier_to_cool_after_days_since_modification_greater_than = 30

        # Move to archive after 90 days
        tier_to_archive_after_days_since_modification_greater_than = 90

        # Delete after 7 years
        delete_after_days_since_modification_greater_than = 2555
      }

      snapshot {
        delete_after_days_since_creation_greater_than = 90
      }
    }
  }
  # Rule 2: Delete temporary files
  rule {
    name    = "cleanup-temp"
    enabled = true

    filters {
      prefix_match = ["temp/"]
      blob_types   = ["blockBlob"]
    }

    actions {
      base_blob {
        delete_after_days_since_modification_greater_than = 7
      }
    }
  }

  # Rule 3: Delete old backups
  rule {
    name    = "cleanup-old-backups"
    enabled = true

    filters {
      prefix_match = ["database-backups/"]
      blob_types   = ["blockBlob"]
    }

    actions {
      base_blob {
        # Keep backups for 30 days, then delete
        delete_after_days_since_modification_greater_than = 30
      }
    }
  }
}
```

How lifecycle works:

```
Day 1: Upload product-images/laptop-001.jpg to Hot tier
       Cost: $0.018/GB/month

Day 30: Automatically moved to Cool tier
        Cost: $0.01/GB/month (44% cheaper)
        Access time: Slightly slower

Day 90: Automatically moved to Archive tier
        Cost: $0.00099/GB/month (95% cheaper!)
        Access time: Hours to retrieve

Day 2555: Automatically deleted
```

5. Managed Disks for VMs

```
# OS disk (created automatically with VM)
resource "azurerm_linux_virtual_machine" "web" {
  name = "vm-web-1"
  # ... other config

  os_disk {
    name                 = "disk-os-web-1"
    caching              = "ReadWrite"
    storage_account_type = "Premium_LRS"  # SSD
    disk_size_gb         = 30
  }
}

# Additional data disk
resource "azurerm_managed_disk" "data" {
  name                 = "disk-data-web-1"
  location             = azurerm_resource_group.storage.location
  resource_group_name  = azurerm_resource_group.storage.name
  storage_account_type = "Premium_LRS"
  create_option        = "Empty"
  disk_size_gb         = 512

  # Encryption
  encryption_settings {
    enabled = true

    disk_encryption_key {
      secret_url      = azurerm_key_vault_secret.disk_key.id
      source_vault_id = azurerm_key_vault.main.id
    }
  }

  tags = {
    Purpose = "Database storage"
  }
}

# Attach disk to VM
resource "azurerm_virtual_machine_data_disk_attachment" "data" {
  managed_disk_id    = azurerm_managed_disk.data.id
  virtual_machine_id = azurerm_linux_virtual_machine.web.id
  lun                = 0  # Logical Unit Number
  caching            = "ReadWrite"
}
```

Disk types comparison:

Type	Use Case	IOPS	Cost
Standard HDD	Cold storage, backups	500	$

Type	Use Case	IOPS	Cost
Standard SSD	Web servers, dev/test	6,000	$
Premium SSD	Databases, production	20,000	$$
Ultra Disk	Mission-critical, SAP	160,000	$$

Using the attached disk in VM:

```
# SSH to VM
ssh azureuser@vm-web-1

# List disks
lsblk
# Output:
# NAME    SIZE TYPE
# sda     30G  disk  (OS disk)
# └─sda1 30G  part
# sdb     512G disk  (Data disk - not formatted!)

# Format the disk
sudo mkfs.ext4 /dev/sdb

# Create mount point
sudo mkdir /data

# Mount disk
sudo mount /dev/sdb /data

# Make permanent
echo "/dev/sdb /data ext4 defaults 0 0" | sudo tee -a /etc/fstab

# Verify
df -h /data
# /dev/sdb        512G   73M  512G   1% /data
```

⭐ **Real Situation #21: Disk Attachment Fails**

Error:

```
Error: creating Data Disk Attachment:
compute.VirtualMachinesClient#CreateOrUpdate:
Failure responding to request: StatusCode=409
Code="OperationNotAllowed"
Message="Cannot attach disk while VM is running"
```

What's happening:

- Azure needs to stop VM to attach disk
- Terraform doesn't automatically stop it
- VM remains running, attachment fails

Fix Option 1: Manually stop VM

```
# Stop VM
az vm stop --resource-group rg-storage --name vm-web-1

# Run Terraform again
terraform apply

# Start VM
az vm start --resource-group rg-storage --name vm-web-1
```

Fix Option 2: Use depends_on and lifecycle (better)

```
resource "azurerm_managed_disk" "data" {
  name = "disk-data-web-1"
  # ... config

  lifecycle {
    # Create disk before attaching
    create_before_destroy = true
  }
}

resource "azurerm_virtual_machine_data_disk_attachment" "data" {
  managed_disk_id    = azurerm_managed_disk.data.id
  virtual_machine_id = azurerm_linux_virtual_machine.web.id
  lun                = 0
  caching            = "ReadWrite"

  # Wait for VM to be fully ready
  depends_on = [azurerm_linux_virtual_machine.web]
}
```

6. Securing Storage with Private Endpoints

```
# Private DNS zone for blob storage
resource "azurerm_private_dns_zone" "blob" {
  name                = "privatelink.blob.core.windows.net"
  resource_group_name = azurerm_resource_group.storage.name
}

# Link DNS zone to VNet
resource "azurerm_private_dns_zone_virtual_network_link" "blob" {
  name                  = "blob-dns-link"
  resource_group_name   = azurerm_resource_group.storage.name
  private_dns_zone_name = azurerm_private_dns_zone.blob.name
  virtual_network_id    = azurerm_virtual_network.main.id
  registration_enabled  = false
}

# Private endpoint for blob storage
resource "azurerm_private_endpoint" "blob" {
  name                = "pe-blob"
  location            = azurerm_resource_group.storage.location
  resource_group_name = azurerm_resource_group.storage.name
  subnet_id           = azurerm_subnet.storage.id

  private_service_connection {
    name                           = "blob-connection"
    private_connection_resource_id = azurerm_storage_account.ecommerce.id
    subresource_names              = ["blob"]
    is_manual_connection           = false
  }

  private_dns_zone_group {
    name                 = "blob-dns-group"
    private_dns_zone_ids = [azurerm_private_dns_zone.blob.id]
  }
}

# Subnet for private endpoints
resource "azurerm_subnet" "storage" {
  name                 = "subnet-storage"
  resource_group_name  = azurerm_resource_group.network.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes     = ["10.0.3.0/24"]

  # Required for private endpoints
  private_endpoint_network_policies_enabled = true
}
```

How private endpoints work:

Before private endpoint:

```
VM in Azure → Internet → Storage Account Public Endpoint
(10.0.1.4)              (stecommerce12ab34cd.blob.core.windows.net)
                             ↓
                    Public IP: 52.239.X.X
```

After private endpoint:

```
VM in Azure → Private Endpoint → Storage Account
(10.0.1.4)    (10.0.3.5)         (No public access!)

DNS resolution:
stecommerce12ab34cd.blob.core.windows.net → 10.0.3.5 (private IP)
```

Benefits:

- ✓ No public internet exposure
- ✓ Traffic stays within Azure backbone
- ✓ Lower latency

- ✓ Higher security
- ✓ No data exfiltration risk

🚨 Chapter 6: 20-Star Troubleshooting Situations

⭐⭐⭐⭐⭐ Situation #22: State File Corruption

Disaster scenario:

```
Power outage during terraform apply
State file corrupted
Terraform can't read state
Infrastructure exists but Terraform doesn't know
```

Symptoms:

```
terraform plan
# Error: Failed to load state: state snapshot was created with newer version
# Or: Error parsing state file: unexpected EOF
```

Recovery steps:

Step 1: Don't panic and don't run terraform apply!

Step 2: Check for state backup

```
ls -la
# terraform.tfstate
# terraform.tfstate.backup  ← Previous good state

# Copy backup
cp terraform.tfstate.backup terraform.tfstate

# Try again
terraform plan  # Should work now
```

Step 3: If using remote state (Azure Blob)

```
# Azure automatically versions blobs
az storage blob list \
  --account-name tfstatestorage \
  --container-name tfstate \
  --prefix terraform.tfstate

# Download previous version
az storage blob download \
  --account-name tfstatestorage \
  --container-name tfstate \
  --name terraform.tfstate \
  --version-id "2024-09-30T10:30:00Z" \
  --file terraform.tfstate.recovered
```

Step 4: If no backup exists (nuclear option)

```
# Remove corrupted state
mv terraform.tfstate terraform.tfstate.corrupted

# Rebuild state by importing
terraform import azurerm_resource_group.main /subscriptions/.../resourceGroups/rg-prod
terraform import azurerm_virtual_network.main /subscriptions/.../virtualNetworks/vnet-prod
# ... import all resources (tedious but necessary)
```

Prevention:

```
# Always use remote state with versioning
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate"
    storage_account_name = "tfstate"
    container_name       = "tfstate"
    key                  = "prod.tfstate"
    # Versioning enabled automatically on Azure Blob
  }
}
```

⭐⭐⭐⭐⭐ Situation #23: Multiple Team Members Applied Simultaneously

Scenario:

```
Person A: terraform apply (10:00 AM)
Person B: terraform apply (10:00 AM)
Both start at same time without locking
```

What happens:

```
Person A creates: VM-1, VM-2
Person B creates: VM-1 (conflict!), VM-3

State file chaos:
- Person A's state: VM-1, VM-2
- Person B's state: VM-1, VM-3
- Reality in Azure: VM-1 (from A), VM-2, VM-3
- But Person B's VM-1 failed, so their state is wrong
```

Symptoms:

```
# Person A's next plan
terraform plan
# Shows: No changes (correct)

# Person B's next plan
terraform plan
# Shows: Want to create VM-1 again (wrong!)
```

Fix:

Step 1: Enable state locking

```
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate"
    storage_account_name = "tfstate"
    container_name       = "tfstate"
    key                  = "prod.tfstate"
    # State locking automatic with Azure Blob
  }
}

# Reinitialize
terraform init -migrate-state
```

Step 2: Reconcile state with reality

```
# Person B: Refresh state from Azure
terraform refresh

# If state still inconsistent, import missing resources
terraform import azurerm_virtual_machine.vm[1] /subscriptions/.../virtualMachines/vm-2
```

Step 3: Implement team workflow

```
# Always pull latest before applying
git pull

# Use workspaces for different environments
terraform workspace new dev
terraform workspace select dev

# Or use separate state files
terraform apply -state="dev.tfstate"
```

⭐⭐⭐⭐⭐ Situation #24: Accidental Destroy of Production

Horror story:

```
# Developer meant to destroy dev environment
cd ~/terraform/dev
terraform destroy  # Oops, wrong directory!

# Actually in production directory
Current directory: ~/terraform/production
# 50 production resources destroyed! 💀
```

Immediate actions:

Step 1: Stop the destroy if still running

```
# Kill terraform process
pkill -9 terraform

# Check what's left
terraform show
```

Step 2: Check what was destroyed

```
# Look at state backup
terraform show terraform.tfstate.backup > before.txt
terraform show > after.txt
diff before.txt after.txt  # See what's missing
```

Step 3: Restore critical resources first

```
# Priority order: Data → Network → Compute

# 1. Restore databases (MOST CRITICAL)
terraform import azurerm_sql_database.main /subscriptions/.../databases/proddb

# 2. Restore network infrastructure
terraform import azurerm_virtual_network.main /subscriptions/.../virtualNetworks/vnet-prod

# 3. Restore VMs
terraform import azurerm_linux_virtual_machine.web /subscriptions/.../virtualMachines/vm-
web-1

# Run apply to restore everything
terraform apply
```

Step 4: Restore data from backups

```
# Restore database from latest backup
az sql db restore \
  --dest-database proddb-restored \
  --server prod-sql-server \
  --resource-group rg-prod \
  --time "2024-09-30T09:00:00Z"  # Before disaster
```

Prevention measures:

```
# 1. Prevent destroy on critical resources
resource "azurerm_sql_database" "main" {
  name = "proddb"
  # ... config

  lifecycle {
    prevent_destroy = true  # Terraform will refuse to destroy
  }
}

# 2. Use workspace-specific naming
resource "azurerm_resource_group" "main" {
  name     = "rg-${terraform.workspace}"  # rg-dev, rg-prod
  location = "East US"
}

# 3. Require approval for production
# In CI/CD pipeline
terraform plan -out=tfplan
# Manual approval step
terraform apply tfplan

# 4. Use Azure locks
resource "azurerm_management_lock" "production" {
  name       = "production-lock"
  scope      = azurerm_resource_group.main.id
  lock_level = "CanNotDelete"  # Prevents deletion
  notes      = "Production resources - do not delete"
}
```

⭐⭐⭐⭐⭐ Situation #25: Terraform Shows Changes Every Run

Frustrating problem:

```
terraform apply
# Apply complete!

terraform plan
# Shows changes even though nothing changed
# This repeats forever!
```

Example:

```
resource "azurerm_virtual_machine_extension" "custom" {
  name                 = "custom-script"
  virtual_machine_id   = azurerm_linux_virtual_machine.web.id
  publisher            = "Microsoft.Azure.Extensions"
  type                 = "CustomScript"
  type_handler_version = "2.1"

  settings = jsonencode({
    commandToExecute = "apt-get update && apt-get install -y nginx"
  })
}

# Every terraform plan shows:
  ~ settings = jsonencode(
      ~ {
          ~ commandToExecute = "apt-get update && apt-get install -y nginx" -> (known after
apply)
        }
    )
```

Root causes:

Cause 1: Provider returns data in different format

```
# You set: "apt-get update && apt-get install -y nginx"
# Azure returns: base64 encoded version
# Terraform sees difference, wants to update
```

Fix for Cause 1:

```
resource "azurerm_virtual_machine_extension" "custom" {
  # ... other config

  settings = jsonencode({
    commandToExecute = "apt-get update && apt-get install -y nginx"
  })

  lifecycle {
    ignore_changes = [settings]  # Stop checking this attribute
  }
}
```

Cause 2: Dynamic values

```
resource "azurerm_app_service" "web" {
  name = "myapp-${formatdate("YYYYMMDD", timestamp())}"
  # timestamp() changes every run!
  # Terraform always sees new name
}
```

Fix for Cause 2:

```
locals {
  # Calculate once when first created
  app_name = "myapp-${formatdate("YYYYMMDD", plantimestamp())}"
}

resource "azurerm_app_service" "web" {
  name = local.app_name
  # Now stable across runs
}
```

Cause 3: Tags keep changing

```
resource "azurerm_resource_group" "main" {
  name     = "rg-prod"
  location = "East US"

  tags = {
    Environment = "Production"
    ManagedBy   = "Terraform"
    LastUpdated = timestamp()  # PROBLEM!
  }
}
```

Fix for Cause 3:

```
resource "azurerm_resource_group" "main" {
  name     = "rg-prod"
  location = "East US"
  tags = {
    Environment = "Production"
    ManagedBy   = "Terraform"
    # Remove timestamp
  }

  lifecycle {
    ignore_changes = [tags["LastUpdated"]]  # If tag added manually
  }
}
```

⭐⭐⭐⭐⭐ Situation #26: Terraform Hangs During Apply

Symptoms:

```
terraform apply

azurerm_linux_virtual_machine.web: Creating...
azurerm_linux_virtual_machine.web: Still creating... [5m0s elapsed]
azurerm_linux_virtual_machine.web: Still creating... [10m0s elapsed]
azurerm_linux_virtual_machine.web: Still creating... [15m0s elapsed]
# Hangs forever, never completes
```

Common causes:

Cause 1: Azure API timeout

```
Azure backend is slow or overloaded
Operation actually completed but Terraform didn't get response
```

Diagnosis:

```
# Check Azure Portal - is VM actually created?
az vm show --resource-group rg-prod --name vm-web-1
# If exists: Terraform lost track

# Check Terraform state
terraform show | grep vm-web-1
# If not in state: Terraform doesn't know it exists
```

Fix:

```
# Cancel terraform (Ctrl+C)
^C

# Import the created resource
terraform import azurerm_linux_virtual_machine.web /subscriptions/.../virtualMachines/vm-
web-1

# Continue
terraform apply
```

Cause 2: Dependency deadlock

```
resource "azurerm_network_interface" "web" {
  name = "nic-web"
  # ... config

  # Circular dependency!
  depends_on = [azurerm_linux_virtual_machine.web]
}

resource "azurerm_linux_virtual_machine" "web" {
  name                  = "vm-web"
  network_interface_ids = [azurerm_network_interface.web.id]
  # Needs NIC, but NIC waits for VM!
}
```

Fix:

```
# Remove incorrect depends_on
resource "azurerm_network_interface" "web" {
  name = "nic-web"
  # ... config
  # No depends_on - Terraform figures it out
}

resource "azurerm_linux_virtual_machine" "web" {
  name                  = "vm-web"
  network_interface_ids = [azurerm_network_interface.web.id]
  # Implicit dependency - correct!
}
```

Cause 3: Custom script never completes

```
resource "azurerm_linux_virtual_machine" "web" {
  # ... config

  custom_data = base64encode(<<-EOF
    #!/bin/bash
    apt-get update
    apt-get install -y some-package-that-does-not-exist
    # This fails, VM boots but script never finishes
    # Terraform waits forever for "success" signal
  EOF
  )
}
```

Fix:

```
resource "azurerm_linux_virtual_machine" "web" {
  # ... config

  custom_data = base64encode(<<-EOF
    #!/bin/bash
    set -e  # Exit on error
    apt-get update || exit 1
    apt-get install -y nginx || exit 1
    systemctl start nginx || exit 1

    # Signal completion
    touch /tmp/cloud-init-complete
  EOF
  )

  # Don't wait for custom_data to complete
  # (Advanced: use remote-exec provisioner with timeout instead)
}
```

📖 Chapter 7: Best Practices Summary

1. Always Use Version Control

```bash
# Initialize git repository
git init
echo ".terraform/" >> .gitignore
echo "*.tfstate" >> .gitignore
echo "*.tfstate.backup" >> .gitignore
echo "terraform.tfvars" >> .gitignore  # Contains secrets

git add .
git commit -m "Initial Terraform configuration"
```

2. State Management Golden Rules

- ✅ Use remote state (Azure Blob, S3, Terraform Cloud)
- ✅ Enable state locking
- ✅ Never edit state files manually
- ✅ Use separate state files per environment
- ✅ Regular state backups

3. Resource Naming Conventions

```
locals {
  # Consistent naming across all resources
  prefix = "${var.project}-${var.environment}"

  common_tags = {
    Project     = var.project
    Environment = var.environment
    ManagedBy   = "Terraform"
    Owner       = var.owner
    CostCenter  = var.cost_center
  }
}

resource "azurerm_resource_group" "main" {
  name     = "rg-${local.prefix}"
  location = var.location
  tags     = local.common_tags
}
```

4. Security Checklist

- ✅ Never commit secrets to git
- ✅ Use Key Vault for sensitive data
- ✅ Enable encryption at rest
- ✅ Use managed identities instead of passwords
- ✅ Implement network security groups
- ✅ Use private endpoints for Azure services
- ✅ Enable Azure Security Center recommendations

5. Testing Your Infrastructure Code

```
# Use terraform validate
terraform validate

# Format check
terraform fmt -check -recursive
```

```
# Plan before apply (ALWAYS)
terraform plan -out=tfplan

# Review plan carefully
terraform show tfplan

# Then apply
terraform apply tfplan
```

6. Documentation Standards

```
# Every variable should have description
variable "vm_count" {
  description = "Number of web server VMs to create. Must be between 1 and 10 for proper
load balancing."
  type        = number
  default     = 3
}

# Every output should have description
output "load_balancer_ip" {
  description = "Public IP address of the load balancer. Use this to access the
application."
  value       = azurerm_public_ip.lb.ip_address
}

# Complex resources should have comments
resource "azurerm_network_security_group" "web" {
  name                = "nsg-web"
  location            = var.location
  resource_group_name = azurerm_resource_group.main.name

  # Allow HTTP from internet (needed for public website)
  security_rule {
    name                       = "AllowHTTP"
    priority                   = 100
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "80"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }
}
```

🎓 Chapter 8: Advanced Troubleshooting Scenarios

⭐⭐⭐⭐⭐ Situation #27: Terraform Drift Detection

Problem: Manual changes in Azure Portal

```
Day 1: Terraform creates VM with size Standard_B2s
Day 5: Someone manually resizes VM to Standard_D4s_v3 in Portal
Day 10: Terraform apply tries to resize back to B2s (destroys data!)
```

Detection:

```
# Run plan to detect drift
terraform plan

# Shows:
  ~ resource "azurerm_linux_virtual_machine" "web" {
      ~ size = "Standard_D4s_v3" -> "Standard_B2s"  # forces replacement
    }

# This means: Current state (D4s) will change to desired state (B2s)
```

Decision tree:

Option 1: Accept manual change (update Terraform to match reality)

```
resource "azurerm_linux_virtual_machine" "web" {
  name = "vm-web-1"
  size = "Standard_D4s_v3"  # Update code to match reality
  # ... rest of config
}

terraform apply  # Now no changes needed
```

Option 2: Revert manual change (enforce Terraform configuration)

```
# VM will be destroyed and recreated!
# Backup data first!

terraform apply  # Enforces B2s size
```

Option 3: Use lifecycle to allow manual changes

```
resource "azurerm_linux_virtual_machine" "web" {
  name = "vm-web-1"
  size = "Standard_B2s"
  # ... rest of config

  lifecycle {
    ignore_changes = [size]  # Allow manual size changes
  }
}

# Now Terraform won't try to change size back
```

Prevention - Implement drift detection in CI/CD:

```bash
#!/bin/bash
# drift-detection.sh

# Run daily
terraform plan -detailed-exitcode

# Exit codes:
# 0 = No changes (good)
# 1 = Error (alert team)
# 2 = Changes detected (drift alert!)

if [ $? -eq 2 ]; then
    echo "DRIFT DETECTED! Manual changes found in infrastructure."
    # Send alert to Slack/Teams
    curl -X POST https://hooks.slack.com/... \
      -d '{"text":"Terraform drift detected in production!"}'
fi
```

⭐⭐⭐⭐⭐ Situation #28: Dependency Hell

Complex scenario:

```
Load Balancer needs:
  → Backend Pool needs:
    → Network Interface needs:
      → Subnet needs:
        → Virtual Network needs:
          → Resource Group

Creating in wrong order = errors!
```

Problem code:

```
# Terraform tries to create these in parallel
# But dependencies are not explicit

resource "azurerm_lb_backend_address_pool" "web" {
  loadbalancer_id = azurerm_lb.web.id
  name            = "BackendPool"
}

resource "azurerm_lb" "web" {
  name                = "lb-web"
  resource_group_name = azurerm_resource_group.main.name
  # ... other config

  # Implicit dependency on resource group ✓
  # But what if resource group not ready?
}

resource "azurerm_network_interface_backend_address_pool_association" "web" {
  network_interface_id    = azurerm_network_interface.web.id
  ip_configuration_name   = "internal"
  backend_address_pool_id = azurerm_lb_backend_address_pool.web.id
}
```

Error:

```
Error: Error waiting for Network Interface "nic-web" to finish updating:
Code="BackendAddressPoolNotFound"
Message="Backend address pool does not exist yet"
```

What went wrong:

```
Timeline:
[0:00] Resource Group created ✓
[0:05] Load Balancer starts creating...
[0:05] Backend Pool tries to create (load balancer not ready!) ✗
[0:06] NIC Association fails (backend pool doesn't exist) ✗
```

Fix with explicit dependencies:

```
resource "azurerm_resource_group" "main" {
  name     = "rg-web"
  location = "East US"
}

resource "azurerm_virtual_network" "main" {
  name                = "vnet-web"
  resource_group_name = azurerm_resource_group.main.name
  # ... config

  depends_on = [azurerm_resource_group.main]
}

resource "azurerm_subnet" "web" {
  name                 = "subnet-web"
  virtual_network_name = azurerm_virtual_network.main.name
  # ... config

  depends_on = [azurerm_virtual_network.main]
}

resource "azurerm_network_interface" "web" {
  name      = "nic-web"
  subnet_id = azurerm_subnet.web.id
  # ... config

  depends_on = [azurerm_subnet.web]
}

resource "azurerm_lb" "web" {
  name                = "lb-web"
  resource_group_name = azurerm_resource_group.main.name
  # ... config

  depends_on = [azurerm_resource_group.main]
}

resource "azurerm_lb_backend_address_pool" "web" {
  loadbalancer_id = azurerm_lb.web.id
  name            = "BackendPool"

  depends_on = [azurerm_lb.web]  # Explicit: wait for LB
}

resource "azurerm_network_interface_backend_address_pool_association" "web" {
  network_interface_id    = azurerm_network_interface.web.id
  ip_configuration_name   = "internal"
  backend_address_pool_id = azurerm_lb_backend_address_pool.web.id

  # Wait for both NIC and backend pool
  depends_on = [
    azurerm_network_interface.web,
    azurerm_lb_backend_address_pool.web
  ]
}
```

Better approach - Use modules:

```
# modules/networking/main.tf
resource "azurerm_virtual_network" "main" {
  # ... config
}

resource "azurerm_subnet" "web" {
  # ... config
}

output "subnet_id" {
  value = azurerm_subnet.web.id
}

# modules/load_balancer/main.tf
variable "subnet_id" {
  description = "Subnet ID from networking module"
}

resource "azurerm_lb" "main" {
  # ... config
}

# main.tf
module "networking" {
  source = "./modules/networking"
  # ... variables
}

module "load_balancer" {
  source    = "./modules/load_balancer"
  subnet_id = module.networking.subnet_id  # Automatic dependency!

  depends_on = [module.networking]  # Ensure networking completes first
}
```

⭐⭐⭐⭐⭐ Situation #29: Quota Limits Hit

Error:

```
Error: creating Linux Virtual Machine "vm-web-6":
compute.VirtualMachinesClient#CreateOrUpdate:
Failure sending request: StatusCode=409
Code="OperationNotAllowed"
Message="Operation could not be completed as it results in exceeding approved
Total Regional Cores quota"
```

What this means:

```
Your subscription has limits:
- Total vCPUs per region: 20
- Currently using: 16 vCPUs
- Trying to create: 5 x Standard_D4s_v3 (4 vCPUs each = 20 vCPUs)
- Total needed: 36 vCPUs
- Over limit by: 16 vCPUs ✗
```

Diagnosis:

```
# Check current quota usage
az vm list-usage --location "East US" --output table

# Shows:
# Name                      CurrentValue  Limit
# Total Regional vCPUs    16            20
# Standard D Family vCPUs 8             10
```

Solutions:

Option 1: Request quota increase

```
# Create support ticket in Azure Portal
# Support → New Support Request → Issue Type: Service and Subscription Limits
# Problem Type: Compute-VM (cores-vCPUs) subscription limit increases

# Usually approved in 1-2 business days
# Free for all Azure subscriptions
```

Option 2: Use smaller VMs

```
resource "azurerm_linux_virtual_machine" "web" {
  count = 5
  name  = "vm-web-${count.index + 1}"
  size  = "Standard_B2s"  # 2 vCPUs instead of 4
  # Total: 10 vCPUs (within quota!)
  # ... rest of config
}
```

Option 3: Deploy to different region

```
variable "location" {
  description = "Azure region"
  default     = "West US 2"  # Different region with available quota
}

resource "azurerm_resource_group" "main" {
  name     = "rg-web"
  location = var.location
}
```

Option 4: Clean up unused resources

```
# Find stopped/deallocated VMs still using quota
az vm list --query "[?powerState=='VM deallocated'].{Name:name,
Size:hardwareProfile.vmSize}" --output table

# Delete unused VMs
az vm delete --resource-group rg-old --name vm-old-1 --yes

# Now quota is freed up
terraform apply  # Should work now
```

⭐⭐⭐⭐⭐ Situation #30: Cross-Region Dependencies

Problem:

```
# Resource group in East US
resource "azurerm_resource_group" "main" {
  name     = "rg-web"
  location = "East US"
}

# Trying to create VM in West US
resource "azurerm_linux_virtual_machine" "web" {
  name                = "vm-web-1"
  location            = "West US"  # Different region!
  resource_group_name = azurerm_resource_group.main.name
  # ... rest of config
}
```

Error:

```
Error: creating Linux Virtual Machine "vm-web-1":
Code="InvalidParameter"
Message="The value 'West US' of parameter 'location' is invalid.
Allowed values are: East US"
```

Why this happens:

- Resources must be in same region as resource group (usually)
- Some resources allow cross-region (storage replication, traffic manager)
- Most compute/network resources must be co-located

Fix:

```
# Option 1: Use same location
resource "azurerm_linux_virtual_machine" "web" {
  name                = "vm-web-1"
  location            = azurerm_resource_group.main.location  # Same as RG
  resource_group_name = azurerm_resource_group.main.name
  # ... rest of config
}
```

```
# Option 2: Create separate resource group per region
resource "azurerm_resource_group" "east" {
  name     = "rg-web-eastus"
  location = "East US"
}

resource "azurerm_resource_group" "west" {
  name     = "rg-web-westus"
  location = "West US"
}

resource "azurerm_linux_virtual_machine" "east" {
  name                = "vm-web-east"
  location            = azurerm_resource_group.east.location
  resource_group_name = azurerm_resource_group.east.name
}

resource "azurerm_linux_virtual_machine" "west" {
  name                = "vm-web-west"
  location            = azurerm_resource_group.west.location
  resource_group_name = azurerm_resource_group.west.name
}
```

Multi-region architecture:

```
# Variable for regions
variable "regions" {
  type    = list(string)
  default = ["East US", "West US", "North Europe"]
}

# Create resource group per region
resource "azurerm_resource_group" "regional" {
  count    = length(var.regions)
  name     = "rg-web-${replace(lower(var.regions[count.index]), " ", "-")}"
  location = var.regions[count.index]
}

# Create VMs in each region
resource "azurerm_linux_virtual_machine" "regional" {
  count               = length(var.regions) * 2  # 2 VMs per region
  name                = "vm-${replace(lower(var.regions[count.index % length(var.regions)]),
" ", "-")}-${count.index}"
  location            = azurerm_resource_group.regional[count.index %
length(var.regions)].location
  resource_group_name = azurerm_resource_group.regional[count.index %
length(var.regions)].name
  # ... rest of config
}

# Result:
# East US: vm-east-us-0, vm-east-us-1
# West US: vm-west-us-2, vm-west-us-3
# North Europe: vm-north-europe-4, vm-north-europe-5
```

⭐⭐⭐⭐⭐ Situation #31: Terraform Upgrade Breaks Everything

Scenario:

```
# You have Terraform 1.3.0
terraform version
# Terraform v1.3.0

# Everything works
terraform plan
# Plan: 0 to add, 0 to change, 0 to destroy

# Upgrade Terraform
brew upgrade terraform
# Now v1.7.0

# Suddenly broken
terraform plan
# Error: Unsupported block type
# Error: Invalid function call
```

Common breaking changes:

Breaking change 1: Provider version mismatch

```
# Old .terraform.lock.hcl specified azurerm 2.x
# New Terraform wants azurerm 3.x
# Resource syntax changed between versions
```

Fix:

```
# Option 1: Pin Terraform version
# .terraform-version file
1.3.0

# Use tfenv to manage versions
tfenv install 1.3.0
tfenv use 1.3.0

# Option 2: Upgrade gradually
# 1. Upgrade provider first
terraform init -upgrade

# 2. Fix deprecated syntax
terraform validate

# 3. Test thoroughly
terraform plan

# 4. Update version constraint
```

```
terraform {
  required_version = ">= 1.7.0"
}
```

Breaking change 2: Function syntax changed

```
# Old (Terraform 1.2)
locals {
  subnet_ids = split(",", var.subnet_string)
}

# New (Terraform 1.5+) - same syntax works, but new functions available
locals {
  subnet_ids = split(",", var.subnet_string)
  # New function: startswith
  is_prod = startswith(var.environment, "prod")
}
```

Best practice for upgrades:

```
# 1. Read changelog
https://github.com/hashicorp/terraform/blob/main/CHANGELOG.md

# 2. Test in dev first
cd dev-environment
terraform init -upgrade
terraform plan
terraform apply

# 3. If successful, upgrade staging
cd staging-environment
terraform init -upgrade
terraform plan
terraform apply

# 4. Finally production (with approval)
cd production-environment
terraform init -upgrade
terraform plan -out=tfplan
# Review carefully
terraform apply tfplan

# 5. Update version constraints
terraform {
  required_version = ">= 1.7.0"

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}
```

🎯 Chapter 9: Real-World Workflow

Complete Production Deployment Workflow

```
# Step 1: Initialize new project
mkdir terraform-production
cd terraform-production

# Step 2: Create directory structure
mkdir -p {modules/{networking,compute,storage},environments/{dev,staging,prod}}

# Step 3: Initialize Git
git init
cat > .gitignore <<EOF
.terraform/
*.tfstate
*.tfstate.backup
*.tfvars
.terraform.lock.hcl
EOF

# Step 4: Create main configuration
cat > main.tf <<EOF
terraform {
  required_version = ">= 1.5.0"

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }

  backend "azurerm" {
    resource_group_name  = "tfstate"
    storage_account_name = "tfstate"
    container_name       = "tfstate"
    key                  = "prod.tfstate"
  }
}

provider "azurerm" {
  features {}
}
EOF

# Step 5: Initialize Terraform
terraform init
```

```
# Step 6: Create workspace for environment
terraform workspace new prod
terraform workspace select prod

# Step 7: Plan changes
terraform plan -out=tfplan

# Step 8: Review plan
terraform show tfplan

# Step 9: Apply (with approval)
terraform apply tfplan

# Step 10: Verify deployment
terraform output

# Step 11: Commit to Git
git add .
git commit -m "Initial production deployment"
git push origin main
```

Daily Operations Checklist

Morning routine:

```
# 1. Check for drift
terraform plan

# 2. Review any changes needed
# If changes: investigate why

# 3. Apply approved changes
terraform apply

# 4. Verify health
az vm list --query "[].powerState" --output table
```

Before making changes:

```
# 1. Create feature branch
git checkout -b feature/add-new-vm

# 2. Make changes to .tf files

# 3. Validate syntax
terraform validate

# 4. Format code
terraform fmt -recursive

# 5. Plan changes
terraform plan -out=tfplan
```

```
# 6. Review carefully
terraform show tfplan

# 7. Apply in dev environment first
cd environments/dev
terraform apply

# 8. Test thoroughly

# 9. Apply to staging
cd ../staging
terraform apply

# 10. Test again

# 11. Apply to production (with approval)
cd ../prod
terraform plan -out=tfplan
# Get approval from team lead
terraform apply tfplan

# 12. Merge to main
git add .
git commit -m "Add new VM for increased capacity"
git push origin feature/add-new-vm
# Create pull request
```

🏁 Conclusion

You now have a comprehensive understanding of Terraform! You've learned:

✅ **Core concepts**: Providers, resources, variables, state, outputs
✅ **Azure provisioning**: VMs, networking, storage in detail
✅ **Troubleshooting**: 20+ real-world scenarios and solutions
✅ **Best practices**: Security, workflows, team collaboration
✅ **Advanced topics**: Drift detection, dependencies, upgrades

Key Takeaways

1. **Always use remote state** - Prevents disasters
2. **Plan before apply** - Catch mistakes early
3. **Use version control** - Track all changes
4. **Test in dev first** - Never experiment in production
5. **Document everything** - Future you will thank you
6. **Automate workflows** - CI/CD for infrastructure
7. **Monitor for drift** - Catch manual changes
8. **Keep learning** - Terraform evolves constantly

Next Steps

1. Practice with the examples in this guide
2. Start with a simple project (1 VM, 1 network)
3. Gradually add complexity
4. Join the Terraform community
5. Read official documentation
6. Experiment safely in dev environments

Remember: **Everyone makes mistakes with Terraform. The difference is learning from them!**

Good luck with your infrastructure-as-code journey! 🚀