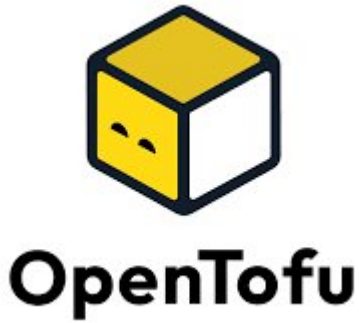


iLGlabs

Redefining how you think, learn, and work.

iLGlabs

By : JP

info@gnugroup.org

www.gnugroup.org

Introduction to OpenTofu {#introduction}

OpenTofu is an open-source Infrastructure as Code (IaC) tool that enables you to define, provision, and manage infrastructure using declarative configuration files. It was forked from Terraform in 2023 and is maintained by the Linux Foundation.

Why OpenTofu?

- **Open Source**: Truly open-source with community governance
- **Declarative**: Describe your desired infrastructure state
- **Cloud Agnostic**: Works with AWS, Azure, GCP, and 100+ providers
- **Version Control**: Infrastructure as code in Git
- **State Management**: Tracks actual vs desired state

Core Concepts and Terminology {#core-concepts}

1. Provider

A provider is a plugin that interacts with a specific cloud platform or service API.

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

2. Resource

A resource is a component of your infrastructure (VM, database, network, etc.).

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

3. Data Source

A data source allows you to fetch information from providers without creating resources.

```hcl
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]
}
```

4. Variable

Variables allow parameterization of your configuration.

hcl

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

5. Output

Outputs display information after infrastructure is created.

```hcl
output "instance_ip" {
  value = aws_instance.example.public_ip
}
```

6. **State**

State is a file that maps your configuration to real-world resources. OpenTofu uses this to track what it manages.

7. **Module**

A module is a container for multiple resources used together. It promotes reusability.

8. **Backend**

A backend determines where state files are stored (local, S3, Azure Blob, etc.).

Providers, Resources & Data Sources

Learning Objectives

By completing this lab, you will:

- Understand provider configuration and version constraints
- Configure multiple providers in a single project
- Create resources with implicit and explicit dependencies
- Use data sources to query existing infrastructure
- Link resources dynamically using data source outputs
- Apply explicit dependency management with `depends_on`

Part 1: Conceptual Overview

1.1 Provider Sources and Versions

What is a Provider?

A provider is a plugin that enables OpenTofu to interact with APIs of cloud platforms, SaaS services, or other infrastructure.

Providers are responsible for:

Understanding API interactions
Exposing resources and data sources
Managing authentication

Provider Configuration Elements:

```hcl
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"         # Registry namespace/name
      version = "~> 5.0"                 # Version constraint
    }
  }
}

provider "aws" {
  region = var.aws_region                # Provider-specific configuration
}
```

Version Constraint Operators:

- `= 1.0.0` - Exact version
- `>= 1.0.0` - Greater than or equal to
- `<= 1.0.0` - Less than or equal to
- `~> 1.0.0` - Allows only rightmost version increment (1.0.x)
- `~> 1.0` - Allows minor version updates (1.x)

1.2 Multiple Provider Usage

Why Use Multiple Providers?

Deploy across multiple cloud platforms

Use utility providers (random, time, null)

Deploy to multiple regions of the same cloud

Integrate with various SaaS services

Provider Aliases:

hcl

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

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

resource "aws_instance" "primary_server" {
  provider = aws.primary
  # ... configuration
}
```

1.3 Resources and Dependencies

Resources are the infrastructure objects you want to create/manage:

EC2 instances
S3 buckets
Security groups
Load balancers

Implicit Dependencies (Automatic):
OpenTofu automatically detects dependencies when you reference one resource in another:

Explicit Dependencies (Manual): Use depends_on when OpenTofu can't detect the relationship:

```
resource "aws_iam_role" "app_role" {
  # ... configuration
}

resource "aws_instance" "app_server" {
  depends_on = [aws_iam_role.app_role]  # Explicit dependency
}
```

1.4 Data Sources

What are Data Sources?
Data sources allow OpenTofu to fetch information about existing infrastructure without managing it.
They're read-only queries.

Common Use Cases:
Look up the latest AMI
Get current AWS account information
Query existing VPC or subnet details
Retrieve DNS information

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

  filter {
    name   = "name"
    values = ["al2023-ami-*-x86_64"]
  }

  owners = ["amazon"]
}

# Reference in resource
resource "aws_instance" "server" {
  ami = data.aws_ami.amazon_linux.id  # Dynamic AMI reference
}
```

```
### Lab Scenario
You'll create a multi-provider infrastructure that:
1. Uses AWS provider for cloud resources
2. Uses random provider for generating unique identifiers
3. Queries existing AWS infrastructure with data sources
4. Creates an EC2 instance with proper dependencies
5. Demonstrates both implicit and explicit dependencies

### Prerequisites Checklist
- [ ] OpenTofu installed (version 1.6+)
- [ ] AWS CLI configured with credentials
- [ ] AWS account with EC2 permissions
- [ ] Text editor (VS Code recommended)
- [ ] Git installed (optional, for version control)
```

Project Structure

```
multi-provider-lab/
├── versions.tf          # Provider requirements and versions
├── variables.tf         # Input variables
├── main.tf              # Primary resource definitions
├── data.tf              # Data source queries
├── outputs.tf           # Output values
└── README.md            # Lab documentation
```

lesson8_multi_provider_lab

OpenTofu : Variables, Functions & Locals

Module Overview

This training module covers advanced OpenTofu variable handling, built-in functions, and dynamic block generation.

We'll learn how to create flexible, reusable infrastructure code that adapts to different requirements without code duplication.

Learning Objectives:

Master complex variable types for sophisticated data structures
Use local values for computed and derived values
Leverage built-in functions for data transformation
Implement dynamic blocks for conditional resource generation
Build a parameterized module with dynamic resource creation

1. Complex Variable Types

1.1 Understanding Type Constraints

OpenTofu supports several complex variable types beyond simple strings and numbers:

Basic Types:

- `string` - Text values
- `number` - Numeric values (integers and floats)
- `bool` - True/false values

Complex Types:

- `list(type)` - Ordered collection of values
- `map(type)` - Key-value pairs with string keys
- `set(type)` - Unordered collection of unique values
- `object({})` - Complex structure with named attributes
- `tuple([])` - Ordered collection with specific types

1.2 Maps

Maps store key-value pairs where keys are strings. Useful for configurations that vary by environment, region, or other categories.

hcl

```hcl
variable "instance_types" {
  description = "EC2 instance types by environment"
  type        = map(string)
  default = {
    dev     = "t3.micro"
    staging = "t3.small"
    prod    = "t3.large"
  }
}

# Usage
resource "aws_instance" "app" {
  instance_type = var.instance_types[var.environment]
  # ...
}
```

Map of Maps (Nested):

hcl

```hcl
variable "regional_settings" {
  description = "Settings by region"
  type        = map(map(string))
  default = {
    us-east-1 = {
      ami           = "ami-0c55b159cbfafe1f0"
      instance_type = "t3.small"
    }
    us-west-2 = {
      ami           = "ami-0d1cd67c26f5fca19"
      instance_type = "t3.medium"
    }
  }
}

# Usage
resource "aws_instance" "regional" {
  ami           = var.regional_settings[var.region]["ami"]
  instance_type = var.regional_settings[var.region]["instance_type"]
}
```

1.3 Objects

Objects define complex structures with specific attribute types. Perfect for grouping related configuration.

hcl

```hcl
variable "database_config" {
  description = "Database configuration"
  type = object({
    engine         = string
    engine_version = string
    instance_class = string
    allocated_storage = number
    multi_az       = bool
    backup_retention = number
  })
  default = {
    engine            = "mysql"
    engine_version    = "8.0"
    instance_class    = "db.t3.micro"
    allocated_storage = 20
    multi_az          = false
    backup_retention  = 7
  }
}

# Usage
resource "aws_db_instance" "main" {
  engine                  = var.database_config.engine
  engine_version          = var.database_config.engine_version
  instance_class          = var.database_config.instance_class
  allocated_storage       = var.database_config.allocated_storage
  multi_az                = var.database_config.multi_az
  backup_retention_period = var.database_config.backup_retention
}
```

List of Objects (Very Common Pattern):

hcl

```hcl
variable "subnets" {
  description = "List of subnet configurations"
  type = list(object({
    name              = string
    cidr_block        = string
    availability_zone = string
    public            = bool
    tags              = map(string)
  }))
  default = [
    {
      name              = "public-subnet-1"
      cidr_block        = "10.0.1.0/24"
      availability_zone = "us-east-1a"
      public            = true
      tags              = { Tier = "Public" }
    },
    {
      name              = "private-subnet-1"
      cidr_block        = "10.0.10.0/24"
      availability_zone = "us-east-1a"
      public            = false
      tags              = { Tier = "Private" }
    }
  ]
}
```

```hcl
# Usage with for_each
resource "aws_subnet" "main" {
  for_each = { for subnet in var.subnets : subnet.name => subnet }

  vpc_id                  = aws_vpc.main.id
  cidr_block              = each.value.cidr_block
  availability_zone       = each.value.availability_zone
  map_public_ip_on_launch = each.value.public

  tags = merge(
    each.value.tags,
    { Name = each.value.name }
  )
}
```

1.4 Tuples

Tuples are ordered collections with specific types for each element. Less common but useful for fixed-structure data.

hcl

```hcl
variable "server_config" {
  description = "Server configuration [name, instance_type, count]"
  type        = tuple([string, string, number])
  default     = ["web-server", "t3.micro", 3]
}

# Usage
locals {
  server_name   = var.server_config[0]
  instance_type = var.server_config[1]
  server_count  = var.server_config[2]
}
```

1.5 Sets

Sets are unordered collections of unique values. Useful when order doesn't matter and duplicates should be eliminated.

hcl

```hcl
variable "allowed_cidr_blocks" {
  description = "Allowed CIDR blocks for security group"
  type        = set(string)
  default     = ["10.0.0.0/8", "172.16.0.0/12", "192.168.0.0/16"]
}

# Usage
resource "aws_security_group_rule" "allow_internal" {
  for_each = var.allowed_cidr_blocks

  type              = "ingress"
  from_port         = 443
  to_port           = 443
  protocol          = "tcp"
  cidr_blocks       = [each.value]
  security_group_id = aws_security_group.main.id
}
```

2. Local Values and Computed Variables

2.1 What are Local Values?

Local values are named expressions that you can reference throughout your configuration.

They help:
Avoid repetition
Compute derived values
Make complex expressions more readable
Combine multiple variable values

Syntax:

```hcl
locals {
  name = expression
}

# Reference with local.name
```

2.2 Common Local Value Patterns

Environment-specific naming:

hcl

```hcl
variable "project_name" {
  type    = string
  default = "myapp"
}

variable "environment" {
  type    = string
  default = "dev"
}

locals {
  # Compute a common prefix for all resources
  name_prefix = "${var.project_name}-${var.environment}"

  # Common tags applied to all resources
  common_tags = {
    Project     = var.project_name
    Environment = var.environment
    ManagedBy   = "OpenTofu"
    CreatedAt   = timestamp()
  }
}
```

```hcl
# Usage
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"

  tags = merge(
    local.common_tags,
    {
      Name = "${local.name_prefix}-vpc"
    }
  )
}
```

Computing CIDR blocks:

hcl

```hcl
variable "vpc_cidr" {
  type    = string
  default = "10.0.0.0/16"
}

locals {
  # Split VPC CIDR into subnet ranges
  subnet_count    = 3
  subnet_newbits  = 8  # Creates /24 subnets from /16

  public_subnet_cidrs = [
    for i in range(local.subnet_count) :
    cidrsubnet(var.vpc_cidr, local.subnet_newbits, i)
  ]

  private_subnet_cidrs = [
    for i in range(local.subnet_count) :
    cidrsubnet(var.vpc_cidr, local.subnet_newbits, i + 10)
  ]
}

# Result:
# public_subnet_cidrs  = ["10.0.0.0/24", "10.0.1.0/24", "10.0.2.0/24"]
# private_subnet_cidrs = ["10.0.10.0/24", "10.0.11.0/24", "10.0.12.0/24"]
```

Flattening nested structures:

hcl

```hcl
variable "applications" {
  type = map(object({
    subnets = list(string)
    ports   = list(number)
  }))
  default = {
    web = {
      subnets = ["subnet-1", "subnet-2"]
      ports   = [80, 443]
    }
    api = {
      subnets = ["subnet-3"]
      ports   = [8080, 8443]
    }
  }
}
```

```hcl
locals {
  # Flatten to create one security group rule per app per port
  security_rules = flatten([
    for app_name, app in var.applications : [
      for port in app.ports : {
        app_name = app_name
        port     = port
      }
    ]
  ])
}

# Result:
# [
#   {app_name = "web", port = 80},
#   {app_name = "web", port = 443},
#   {app_name = "api", port = 8080},
#   {app_name = "api", port = 8443}
# ]
```

2.3 Local Values vs Variables

Use Variables when:
Values should be configurable by users
Different values needed per environment
Values come from external sources

Use Locals when:
Computing derived values
Complex expressions used multiple times
Internal logic not meant for external configuration

3. Built-in Functions

OpenTofu provides 100+ built-in functions for data transformation.

3.1 Lookup Functions

lookup(map, key, default)

hcl

```hcl
variable "instance_types" {
  type = map(string)
  default = {
    dev  = "t3.micro"
    prod = "t3.large"
  }
}

# Safely lookup with fallback
locals {
  instance_type = lookup(var.instance_types, var.environment, "t3.small")
}

# If var.environment = "staging" (not in map), returns "t3.small"
```

element(list, index)

hcl

```hcl
variable "availability_zones" {
  type    = list(string)
  default = ["us-east-1a", "us-east-1b", "us-east-1c"]
}

# Cycle through AZs for multiple resources
resource "aws_subnet" "example" {
  count             = 6
  availability_zone = element(var.availability_zones, count.index)
  # Creates 2 subnets per AZ: 1a, 1b, 1c, 1a, 1b, 1c
}
```

index(list, value)

hcl

```hcl
locals {
  environments = ["dev", "staging", "prod"]
  env_index    = index(local.environments, var.environment)
  # Returns 0 for dev, 1 for staging, 2 for prod
}
```

3.2 File Functions

file(path)

hcl

```hcl
# Read file contents as string
resource "aws_instance" "web" {
  user_data = file("${path.module}/scripts/init.sh")
}
```

templatefile(path, vars)

hcl

```hcl
# Read and interpolate template
resource "aws_instance" "web" {
  user_data = templatefile("${path.module}/templates/init.sh.tpl", {
    db_host     = aws_db_instance.main.endpoint
    environment = var.environment
    app_version = var.app_version
  })
}

# init.sh.tpl:
# #!/bin/bash
# export DB_HOST="${db_host}"
# export ENVIRONMENT="${environment}"
# export APP_VERSION="${app_version}"
```

filebase64(path)

hcl

```hcl
# Read file as base64 (useful for binary files)
resource "aws_s3_object" "image" {
  bucket  = aws_s3_bucket.assets.id
  key     = "logo.png"
  content_base64 = filebase64("${path.module}/assets/logo.png")
}
```

3.3 String Functions

format(spec, values...)

hcl

```hcl
locals {
  instance_name = format("%s-%s-instance-%02d",
    var.project_name,
    var.environment,
    count.index + 1
  )
  # Result: "myapp-prod-instance-01"
}
```

join(separator, list)

hcl

```hcl
variable "name_parts" {
  type    = list(string)
  default = ["myapp", "web", "server"]
}

locals {
  full_name = join("-", var.name_parts)
  # Result: "myapp-web-server"
}
```

split(separator, string)

hcl

```hcl
locals {
  db_endpoint = "mydb.abc123.us-east-1.rds.amazonaws.com:3306"
  db_parts    = split(":", local.db_endpoint)
  db_host     = local.db_parts[0]
  db_port     = local.db_parts[1]
}
```

replace(string, search, replace)

```hcl
locals {
  sanitized_name = replace(var.project_name, "_", "-")
  # Replaces underscores with hyphens
}
```

3.4 Numeric Functions

min(numbers...) / max(numbers...)

hcl

```hcl
variable "instance_counts" {
  type = map(number)
  default = {
    web = 3
    api = 5
    worker = 2
  }
}

locals {
  min_instances = min(var.instance_counts["web"],
                      var.instance_counts["api"],
                      var.instance_counts["worker"])
  max_instances = max(var.instance_counts...)  # Splat syntax
}
```

ceil(number) / floor(number)

hcl

```hcl
locals {
  storage_gb_needed = 47.8
  storage_rounded   = ceil(local.storage_gb_needed)  # 48
}
```

3.5 Collection Functions

merge(maps...)

```hcl
locals {
  default_tags = {
    ManagedBy = "OpenTofu"
    Project   = var.project_name
  }

  environment_tags = {
    Environment = var.environment
  }

  all_tags = merge(
    local.default_tags,
    local.environment_tags,
    var.custom_tags
  )
}
```

concat(lists...)

```hcl
locals {
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
  private_subnets = ["10.0.10.0/24", "10.0.11.0/24"]
  all_subnets     = concat(local.public_subnets, local.private_subnets)
}
```

flatten(list_of_lists)

hcl

```hcl
locals {
  nested_cidrs = [
    ["10.0.1.0/24", "10.0.2.0/24"],
    ["10.0.10.0/24", "10.0.11.0/24"]
  ]
  flat_cidrs = flatten(local.nested_cidrs)
  # ["10.0.1.0/24", "10.0.2.0/24", "10.0.10.0/24", "10.0.11.0/24"]
}
```

keys(map) / values(map)

hcl

```hcl
variable "regions" {
  type = map(string)
  default = {
    us-east-1 = "N. Virginia"
    us-west-2 = "Oregon"
  }
}

locals {
  region_codes = keys(var.regions)        # ["us-east-1", "us-west-2"]
  region_names = values(var.regions)      # ["N. Virginia", "Oregon"]
}
```

3.6 Conditional Functions

coalesce(values...)

```hcl
# Returns first non-null, non-empty value
locals {
  instance_type = coalesce(
    var.instance_type_override,
    var.default_instance_type,
    "t3.micro"
  )
  # Uses override if set, otherwise default, otherwise t3.micro
}
```

try(expressions...)

hcl

```hcl
# Returns first expression that doesn't error
locals {
  ami_id = try(
    var.custom_ami_id,
    data.aws_ami.amazon_linux.id,
    "ami-default"
  )
}
```

can(expression)

hcl

```hcl
# Returns true if expression evaluates without error
locals {
  has_custom_ami = can(regex("^ami-", var.ami_id))
}
```

3.7 Network Functions

cidrsubnet(prefix, newbits, netnum)

---

hcl

```hcl
# Split VPC CIDR into smaller subnets
locals {
  vpc_cidr = "10.0.0.0/16"

  # Create /24 subnets
  subnet_1 = cidrsubnet(local.vpc_cidr, 8, 0)   # 10.0.0.0/24
  subnet_2 = cidrsubnet(local.vpc_cidr, 8, 1)   # 10.0.1.0/24
  subnet_3 = cidrsubnet(local.vpc_cidr, 8, 10)  # 10.0.10.0/24
}
```

cidrhost(prefix, hostnum)

hcl

```hcl
# Get specific IP from CIDR block
locals {
  subnet = "10.0.1.0/24"
  first_ip = cidrhost(local.subnet, 1)   # 10.0.1.1
  tenth_ip = cidrhost(local.subnet, 10)  # 10.0.1.10
}
```

4. Dynamic and Conditional Blocks

4.1 Dynamic Blocks

Dynamic blocks allow you to generate nested blocks within resources based on variable content. Essential for creating flexible, reusable modules.

Basic Syntax:

hcl

```hcl
dynamic "block_name" {
  for_each = var.list_or_map
  content {
    # Block attributes using each.value or each.key
  }
}
```

Example: Dynamic Ingress Rules

hcl

```hcl
variable "ingress_rules" {
  description = "List of ingress rules"
  type = list(object({
    from_port   = number
    to_port     = number
    protocol    = string
    cidr_blocks = list(string)
    description = string
  }))
  default = [
    {
      from_port   = 80
      to_port     = 80
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
      description = "HTTP from anywhere"
    },
    {
      from_port   = 443
      to_port     = 443
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
      description = "HTTPS from anywhere"
    }
  ]
}
```

```hcl
resource "aws_security_group" "web" {
  name        = "web-sg"
  vpc_id      = aws_vpc.main.id

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

Example: Dynamic Tags

hcl

```hcl
variable "enable_detailed_monitoring" {
  type    = bool
  default = false
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"

  dynamic "tag_specifications" {
    for_each = var.enable_detailed_monitoring ? [1] : []
    content {
      resource_type = "instance"
      tags = {
        DetailedMonitoring = "enabled"
      }
    }
  }
}
```

4.2 Conditional Blocks with for_each

Use empty maps or sets to conditionally create blocks:

hcl

```hcl
variable "enable_backup" {
  type    = bool
  default = true
}

resource "aws_db_instance" "main" {
  # ... other config ...

  dynamic "backup_configuration" {
    for_each = var.enable_backup ? [1] : []
    content {
      backup_retention_period = 7
      backup_window           = "03:00-04:00"
      maintenance_window      = "mon:04:00-mon:05:00"
    }
  }
}
```

4.3 Nested Dynamic Blocks

You can nest dynamic blocks for complex structures:

hcl

```hcl
variable "load_balancer_listeners" {
  type = list(object({
    port     = number
    protocol = string
    rules = list(object({
      priority = number
      host     = string
      path     = string
    }))
  }))
}

resource "aws_lb_listener" "main" {
  for_each = { for listener in var.load_balancer_listeners : listener.port => listener }

  load_balancer_arn = aws_lb.main.arn
  port              = each.value.port
  protocol          = each.value.protocol

  dynamic "default_action" {
    for_each = each.value.rules
    content {
      type = "forward"

      dynamic "forward" {
        for_each = [default_action.value]
        content {
          target_group {
            arn    = aws_lb_target_group.main.arn
            weight = 100
          }
        }
      }
    }
  }
}
```

4.4 Conditional Resource Creation

Using count:

hcl

```hcl
variable "create_nat_gateway" {
  type    = bool
  default = true
}

resource "aws_nat_gateway" "main" {
  count = var.create_nat_gateway ? 1 : 0

  allocation_id = aws_eip.nat[0].id
  subnet_id     = aws_subnet.public[0].id
}

# Reference with [0] when used
resource "aws_route" "private_nat" {
  count = var.create_nat_gateway ? 1 : 0

  route_table_id         = aws_route_table.private.id
  destination_cidr_block = "0.0.0.0/0"
  nat_gateway_id         = aws_nat_gateway.main[0].id
}
```

Using for_each:

hcl

```hcl
variable "create_bastion" {
  type    = bool
  default = false
}

resource "aws_instance" "bastion" {
  for_each = var.create_bastion ? { bastion = true } : {}

  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"
  subnet_id     = aws_subnet.public[0].id
}
```

Key Takeaways

1. **Complex Types** enable sophisticated data structures matching real-world infrastructure needs
2. **Local Values** reduce repetition and make complex logic readable
3. **Built-in Functions** provide powerful data transformation without external dependencies
4. **Dynamic Blocks** create flexible modules that adapt to variable inputs
5. **Conditional Logic** allows single codebase to handle multiple scenarios

LAB: Parameterized VPC Module with Dynamic Subnets and Security Groups

Lab Overview

Build a reusable VPC module that dynamically creates subnets across availability zones and generates security groups with flexible rule sets. This lab demonstrates all advanced variable concepts in a real-world scenario.

What You'll Build:

Parameterized VPC module accepting complex variable types
Dynamic subnet creation across multiple AZs
Dynamic security group generation with computed rules
Local values for CIDR calculations
Built-in functions for data transformation

Architecture

```
VPC Module
├── Dynamic Public Subnets (across multiple AZs)
├── Dynamic Private Subnets (across multiple AZs)
├── Dynamic Database Subnets (across multiple AZs)
├── Internet Gateway (conditional)
├── NAT Gateways (conditional, per AZ)
├── Route Tables (dynamic based on subnet tiers)
└── Security Groups (dynamic rules based on application requirements
```

Project Structure

Create this directory structure:

```
lab-vpc-module/
├── modules/
│   └── vpc/
│       ├── main.tf
│       ├── variables.tf
│       ├── outputs.tf
│       ├── locals.tf
│       ├── subnets.tf
│       ├── security-groups.tf
│       └── routing.tf
├── environments/
│   ├── dev/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   ├── terraform.tfvars
│   │   └── versions.tf
│   └── prod/
│       ├── main.tf
│       ├── variables.tf
│       ├── terraform.tfvars
│       └── versions.tf
└── README.md
```

LAB - lesson11_Adv_variables_functions_locals_vpc_modules

OpenTofu Modules and Registries

Learning Objectives

By the end of this training, you will be able to:

Understand the difference between root and child modules
Create reusable, production-ready modules
Use local and remote module sources effectively
Leverage community modules from registries
Publish and version your own modules
Implement proper module versioning strategies
Apply best practices for module design and composition

1. Understanding OpenTofu Modules

What Are Modules?

A **module** is a container for multiple resources that are used together.

Every OpenTofu configuration has at least one module, called the **root module**, which consists of the resources defined in the `.tf` files in the main working directory.

Root vs Child Modules

Root Module:

The main working directory where you run tofu commands
Contains your primary configuration files
Calls child modules to compose infrastructure
Manages state directly

Child Modules:

Reusable components called by the root module
Self-contained with their own resources, variables, and outputs
Can be called multiple times with different configurations
Promote DRY (Don't Repeat Yourself) principle

Why Use Modules?

1. **Reusability**: Write once, use many times
2. **Organization**: Logical grouping of related resources
3. **Encapsulation**: Hide complexity behind simple interfaces
4. **Standardization**: Enforce organizational best practices
5. **Collaboration**: Share infrastructure patterns across teams
6. **Testing**: Isolate and test components independently

2. Creating Reusable Modules

Module Structure Best Practices

A well-structured module typically contains:

```
module-name/
├── README.md           # Documentation
├── main.tf             # Primary resource definitions
├── variables.tf        # Input variables
├── outputs.tf          # Output values
├── versions.tf         # Provider requirements
├── examples/           # Usage examples
│   └── basic/
│       ├── main.tf
│       └── variables.tf
└── tests/              # Validation tests (optional)
```

Key Principles for Module Design

1. Single Responsibility

- Each module should do one thing well
- Focus on a specific infrastructure pattern
- Avoid overly complex, multi-purpose modules

2. Configurable But Opinionated

- Provide sensible defaults
- Allow customization through variables
- Guide users toward best practices

3. Clear Interface

- Well-documented input variables
- Meaningful output values
- Explicit requirements and constraints

4. Composability

- Modules should work well together
- Avoid tight coupling between modules
- Use outputs/inputs for module communication

Example: Simple Module Structure

Root Module (calls the module):

```hcl
module "web_server" {
  source = "./modules/ec2-instance"

  instance_name = "web-server-1"
  instance_type = "t3.micro"
  subnet_id     = aws_subnet.public.id
}
```

Child Module (./modules/ec2-instance/main.tf):

```hcl
resource "aws_instance" "this" {
  ami           = var.ami_id
  instance_type = var.instance_type
  subnet_id     = var.subnet_id

  tags = {
    Name = var.instance_name
  }
}
```

3. Module Sources: Local vs Remote

3. Module Sources: Local vs Remote

Local Modules

Path Syntax:

```hcl
module "local_example" {
  source = "./modules/vpc"        # Relative path
  source = "/absolute/path/vpc"   # Absolute path (not recommended)
}
```

Use Cases:

- Development and testing
- Organization-specific modules not for public us
- Tightly coupled components
- When you need to modify modules frequently

Advantages:

- Fast iteration during development
- No versioning complexity
- Full control over code

Disadvantages:

- No version control
- Must be distributed with root module
- Harder to share across projects

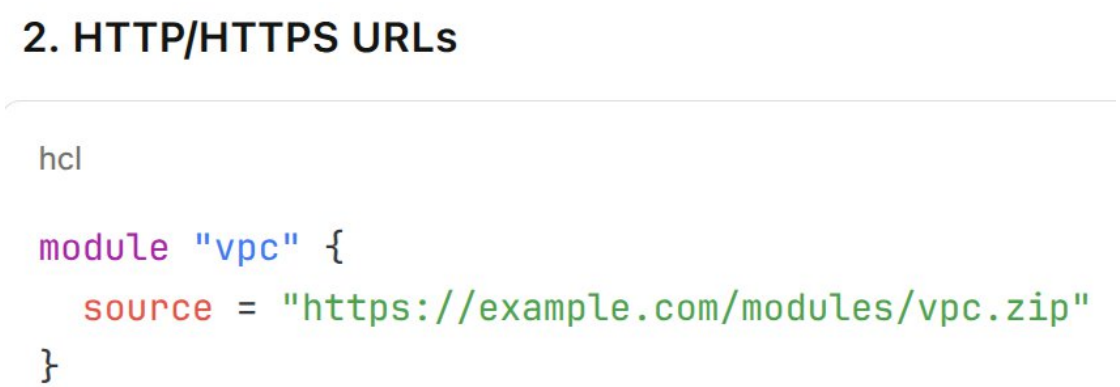
Remote Modules

OpenTofu supports multiple remote sources:

1. Git Repositories

```
# HTTPS
module "vpc" {
  source = "git::https://github.com/example/terraform-aws-vpc.git"
}

# SSH
module "vpc" {
  source = "git::ssh://git@github.com/example/terraform-aws-vpc.git"
}

# Specific branch
module "vpc" {
  source = "git::https://github.com/example/terraform-aws-vpc.git?ref=main"
}

# Specific tag (versioning)
module "vpc" {
  source = "git::https://github.com/example/terraform-aws-vpc.git?ref=v1.2.0"
}

# Specific commit
module "vpc" {
  source = "git::https://github.com/example/terraform-aws-vpc.git?ref=abc123"
}

# Subdirectory
module "vpc" {
  source = "git::https://github.com/example/modules.git//vpc"
}
```

2. HTTP/HTTPS URLs



3. Registry Sources (Terraform Registry)

```hcl
# Terraform Registry format
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.0"
}
```

4. Local Archives

```hcl
module "vpc" {
  source = "./modules/vpc-1.0.0.zip"
}
```

Module Versioning Best Practices

Semantic Versioning (SemVer):

- Format: `MAJOR.MINOR.PATCH` (e.g., `1.4.2`)
- **MAJOR**: Breaking changes
- **MINOR**: New features, backward compatible
- **PATCH**: Bug fixes, backward compatible

Version Constraints:

```hcl
module "vpc" {
  source  = "git::https://github.com/example/vpc.git?ref=v1.2.0"
  # Or for registry modules:
  version = "~> 1.2.0"  # >= 1.2.0, < 1.3.0
  version = ">= 1.2.0"  # Any version >= 1.2.0
  version = "1.2.0"     # Exact version
}
```

4. Using Community Registry Modules

OpenTofu Registry (Library.tf)

Library.tf is the community-driven module registry for OpenTofu, similar to Terraform Registry but specifically for OpenTofu.

Key Features:

- Curated collection of OpenTofu modules
- Version management
- Documentation and examples
- Community contributions
- Compatible with OpenTofu-specific features

Finding Modules on Library.tf

1. Visit https://library.tf
2. Search by provider (AWS, Azure, GCP, etc.)
3. Filter by category (networking, compute, security, etc.)
4. Review module documentation and examples
5. Check version history and compatibility

Basic Usage:

```
module "vpc" {
  source  = "registry.opentofu.org/terraform-aws-modules/vpc/aws"
  version = "5.1.0"

  name = "my-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["us-east-1a", "us-east-1b", "us-east-1c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = false

  tags = {
    Environment = "dev"
    ManagedBy   = "OpenTofu"
  }
}
```

Evaluating Community Modules

Quality Checklist:

- [ ] Well-documented with clear examples
- [ ] Active maintenance (recent commits)
- [ ] Proper versioning scheme
- [ ] Comprehensive input variables
- [ ] Useful output values
- [ ] Tests or validation examples
- [ ] Clear license
- [ ] Community usage and stars

Security Considerations:

- Review source code before use
- Pin to specific versions
- Check for sensitive data handling
- Verify provider requirements
- Test in non-production first

Hands-On Lab: Build and Reuse a VPC Module with Versioning

In this lab, you will:

1. Create a reusable VPC module from scratch
2. Use the module locally in a root configuration
3. Version the module using Git tags
4. Publish the module to GitHub
5. Consume the module with version constraints
6. Iterate on the module and release a new version

Prerequisites:

- OpenTofu installed (1.6.0 or later)
- AWS CLI configured with credentials
- Git installed and configured
- GitHub account (for publishing)
- Text editor or IDE

What You'll Build: A production-ready VPC module that creates:

- VPC with customizable CIDR
- Public and private subnets across multiple AZs
- Internet Gateway
- NAT Gateway (optional)
- Route tables and associations
- Proper tagging and naming

OpenTofu State Management & Remote Backends

Learning Objectives

By the end of this module, you will be able to:

Understand the purpose and structure of OpenTofu state
Compare local and remote state storage approaches
Configure remote backends (S3, GCS, Azure Blob Storage)
Implement state locking using DynamoDB
Import existing infrastructure into state
Manipulate state safely using state commands
Migrate from local to remote state storage

Prerequisites

Basic understanding of OpenTofu resources and providers
AWS account with appropriate permissions
OpenTofu CLI installed (v1.6 or later)
AWS CLI configured with credentials

1. Understanding OpenTofu State

What is State?

OpenTofu state is a JSON file that maps your configuration to real-world resources. It serves as the "source of truth" for your infrastructure.

Key purposes of state:

- **Resource tracking**: Maintains metadata about resources managed by OpenTofu
- **Performance optimization**: Caches resource attributes to avoid constant API calls
- **Dependency mapping**: Tracks relationships between resources
- **Drift detection**: Enables comparison between desired and actual state

State File Structure

A typical state file contains:

```
{
  "version": 4,
  "terraform_version": "1.6.0",
  "serial": 5,
  "lineage": "unique-uuid",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "web",
      "provider": "provider[\"registry.opentofu.org/hashicorp/aws\"]",
      "instances": [...]
    }
  ]
}
```

Important fields:

- `version`: State file format version
- `serial`: Increments with each state modification
- `lineage`: Unique identifier for the state history
- `resources`: Array of all managed resources

The Dangers of State

State files contain sensitive information:

- Resource IDs and ARNs
- IP addresses and DNS names
- Database passwords (if stored in plaintext)
- API keys and secrets

Never commit state files to version control!

2. Local vs Remote State

Local State

How it works:

- State stored in `terraform.tfstate` file in working directory
- Simple, no additional configuration needed
- Suitable for personal projects and learning

Advantages:

- Simple setup, no dependencies
- Fast read/write operations
- No network requirements
- Easy to inspect and debug

Disadvantages:

- No collaboration support (single user only)
- No state locking (risk of concurrent modifications)
- Risk of accidental deletion or corruption
- Difficult to share across teams
- No encryption at rest by default
- State contains sensitive data on local disk

When to use local state:

- Personal learning projects
- Proof of concepts
- Temporary test environments
- Single-user development scenarios

Remote State

How it works:

- State stored in a remote location (S3, GCS, Azure Blob, etc.)
- Multiple team members can access the same state
- Supports state locking to prevent concurrent modifications
- Often includes encryption and versioning

Advantages:

- Team collaboration enabled
- State locking prevents conflicts
- Encryption at rest and in transit
- Versioning and backup capabilities
- Centralized state management
- Audit trail of state changes
- Better security controls

Disadvantages:

- Additional infrastructure required
- Network dependency
- Slightly slower operations
- More complex setup
- Potential costs for storage and API calls

When to use remote state:

- Team environments
- Production infrastructure
- CI/CD pipelines
- Any shared infrastructure

3. Remote Backend Options

S3 Backend (AWS)

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

    # Optional: Use specific profile
    profile = "production"

    # Optional: Server-side encryption
    kms_key_id = "arn:aws:kms:us-east-1:123456789012:key/12345678-1234-1234-123
  }
}
```

Features:

- Encryption at rest using AES-256 or KMS
- Versioning support for state history
- State locking via DynamoDB
- Cost-effective for most use cases
- Integrates with AWS IAM for access control

Requirements:

- S3 bucket with versioning enabled
- DynamoDB table for state locking
- IAM permissions for S3 and DynamoDB

GCS Backend (Google Cloud)

```
terraform {
  backend "gcs" {
    bucket  = "my-tofu-state"
    prefix  = "prod"

    # Optional: Encryption
    encryption_key = "base64-encoded-encryption-key"

    # Optional: Specific credentials
    credentials = "path/to/service-account-key.json"
  }
}
```

Features:

- Automatic encryption at rest
- Object versioning
- Built-in state locking (no additional service needed)
- IAM integration
- Audit logging via Cloud Audit Logs

Requirements:

- GCS bucket
- Service account with appropriate permissions
- Google Cloud SDK configured

Azure Blob Storage Backend

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

    # Optional: Use specific subscription
    subscription_id = "00000000-0000-0000-0000-000000000000"

    # Optional: Use managed identity
    use_msi = true
  }
}
```

Features:

- Encryption at rest enabled by default
- Built-in state locking via blob leases
- Versioning support
- Azure RBAC integration
- Soft delete capability

Requirements:

- Azure Storage Account
- Storage container
- Appropriate Azure AD permissions

Other Backend Options

HTTP Backend:

- Custom REST API endpoints
- Useful for custom solutions
- Requires implementing the backend protocol

Consul Backend:

- HashiCorp Consul key/value store
- Built-in locking
- Good for HashiCorp stack integration

Terraform Cloud/Enterprise:

- Managed remote state
- Built-in VCS integration
- Policy enforcement
- Cost estimation

4. State Locking with DynamoDB

Why State Locking Matters

The Problem: Two team members run `tofu apply` simultaneously:

1. Both read the current state
2. Both make changes
3. Both try to write state
4. One overwrites the other's changes
5. Infrastructure drift and lost changes occur

The Solution: State locking ensures only one operation can modify state at a time.

How DynamoDB Locking Works

1. Before modifying state, OpenTofu attempts to acquire a lock
2. Lock information is written to DynamoDB table
3. If lock exists, operation waits or fails
4. After state update, lock is released
5. Lock includes metadata: who, when, operation ID

DynamoDB Table Setup

Required table structure:

```
resource "aws_dynamodb_table" "tofu_locks" {
  name           = "tofu-state-lock"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "LockID"

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

  tags = {
    Name        = "OpenTofu State Lock Table"
    Environment = "Production"
  }
}
```

Key points:

- Table name referenced in backend configuration
- `LockID` is the primary key (String type)
- PAY_PER_REQUEST is cost-effective for most use cases
- No additional attributes needed

Lock Behavior

Automatic locking:

- `tofu apply` - Locks state during plan and apply
- `tofu destroy` - Locks during destroy operation
- `tofu refresh` - Locks during refresh
- `tofu import` - Locks during import

Force unlock:

```
# If a lock gets stuck (use with extreme caution)
tofu force-unlock <LOCK_ID>
```

Warning: Only force-unlock if you're certain no other process is running!

Handling Lock Conflicts

Common scenarios:

Scenario 1: Legitimate concurrent operation

```
Error: Error acquiring the state lock
Lock Info:
  ID:        abc123-def456-ghi789
  Path:      my-bucket/terraform.tfstate
  Operation: OperationTypeApply
  Who:       john@example.com
  Version:   1.6.0
  Created:   2024-01-15 10:30:45 UTC
```

Resolution: Wait for the other operation to complete.

Scenario 2: Stale lock (process crashed)

bash

```bash
# Verify no process is actually running
# Then force unlock
tofu force-unlock abc123-def456-ghi789
```

5. State Manipulation Commands

Viewing State

List all resources:

```bash
tofu state list
```

Example output:

```
aws_vpc.main
aws_subnet.public[0]
aws_subnet.public[1]
aws_instance.web
```

Show specific resource:

```bash
tofu state show aws_instance.web
```

Show all state details:

```bash
tofu show
```

Importing Existing Resources

Use case: Bring existing infrastructure under OpenTofu management.

Basic import:

```bash
tofu import aws_instance.web i-1234567890abcdef0
```

Import with configuration:

1. Create the resource block in your configuration:

```hcl
resource "aws_instance" "web" {
  # Configuration will be filled in after import
  ami           = "ami-12345678"
  instance_type = "t3.micro"
}
```

2. Import the existing resource:

```bash
tofu import aws_instance.web i-1234567890abcdef0
```

3. Run plan to see any drift:

```bash
tofu plan
```

4. Update configuration to match actual state (eliminate drift)

Import with modules:

```bash
tofu import module.vpc.aws_vpc.main vpc-12345678
```

Import with for_each:

```bash
tofu import 'aws_subnet.public["us-east-1a"]' subnet-12345678
```

Moving Resources in State

Rename a resource:

```bash
tofu state mv aws_instance.web aws_instance.webserver
```

Move to a module:

```bash
tofu state mv aws_instance.web module.web.aws_instance.server
```

Move from a module:

```bash
tofu state mv module.web.aws_instance.server aws_instance.web
```

Move between indexed resources:

```bash
tofu state mv 'aws_subnet.public[0]' 'aws_subnet.public[1]'
```

Common use cases:

- Refactoring module structure
- Renaming resources without recreating them
- Reorganizing resource hierarchies

Removing Resources from State

Remove without destroying:

```bash
tofu state rm aws_instance.old_server
```

What happens:

- Resource removed from state
- Physical resource remains unchanged
- OpenTofu will no longer manage it
- Next apply won't destroy it

Use cases:

- Handing off management to another tool
- Removing resources before deletion
- Cleaning up after failed imports

Replacing Resources

Mark resource for replacement:

```bash
tofu apply -replace=aws_instance.web
```

What happens:

- Resource is destroyed and recreated
- Even if configuration hasn't changed
- Useful for troubleshooting

Pulling and Pushing State

Pull current state:

```bash
tofu state pull > terraform.tfstate.backup
```

Push modified state (DANGEROUS):

```bash
tofu state push terraform.tfstate.modified
```

Warning: Manual state push can corrupt state. Only use in recovery scenarios!

State Command Best Practices

1. **Always backup before manipulation:**

```bash
tofu state pull > backup-$(date +%Y%m%d-%H%M%S).tfstate
```

2. **Use `-dry-run` when available:**

```bash
tofu state mv -dry-run aws_instance.old aws_instance.new
```

3. **Verify changes:**

```bash
tofu plan
```

4. **Use state locking:** Ensure backend has locking enabled
5. **Document changes:** Keep notes on why state was manipulated

6. State Migration Strategies

Migration Planning

Questions to answer:

- Where is state currently stored?
- Where should it be stored?
- Is state locking configured?
- Are there multiple state files to migrate?
- Do team members need access?

Migration Process

Step 1: Backup current state

bash

```bash
cp terraform.tfstate terraform.tfstate.backup
```

Step 2: Create remote backend resources

- S3 bucket
- DynamoDB table
- IAM permissions

Step 3: Add backend configuration

hcl

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

Step 4: Initialize with migration

bash

```bash
tofu init -migrate-state
```

Step 5: Verify migration

bash

```bash
tofu plan
# Should show no changes if migration successful
```

Step 6: Clean up local state (after verification)

bash

```bash
# Move to archive, don't delete immediately
mkdir -p old-state
mv terraform.tfstate* old-state/
```

Partial Backend Configuration

Use case: Different backends per environment without hardcoding values.

backend.hcl:

```hcl
bucket         = "prod-tofu-state"
key            = "prod/terraform.tfstate"
region         = "us-east-1"
encrypt        = true
dynamodb_table = "prod-tofu-locks"
```

Initialize with partial config:

```bash
tofu init -backend-config=backend.hcl
```

Benefits:

- Keep sensitive values out of version control
- Use different configs per environment
- CI/CD friendly approach

7. Security Best Practices

Encryption

S3 Backend:

- Enable default encryption on bucket
- Use KMS for enhanced security
- Enable bucket versioning

Secrets in State:

- Avoid storing secrets in plaintext
- Use AWS Secrets Manager or Vault
- Reference secrets, don't embed them

Access Control

S3 Bucket Policy:

```hcl
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/Tofu-Role"
      },
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject"
      ],
      "Resource": "arn:aws:s3:::my-tofu-state/*"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:role/Tofu-Role"
      },
      "Action": "s3:ListBucket",
      "Resource": "arn:aws:s3:::my-tofu-state"
    }
  ]
}
```

DynamoDB Permissions

Required permissions:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:DeleteItem"
      ],
      "Resource": "arn:aws:dynamodb:*:*:table/tofu-state-lock"
    }
  ]
}
```

Audit and Monitoring

Enable CloudTrail logging:

- Track who accessed state
- Monitor state modifications
- Alert on suspicious activity

S3 Access Logging:

- Log all bucket access
- Review access patterns
- Detect unauthorized access

8. Common Patterns and Anti-Patterns

✅ Good Patterns

Environment Separation:

```
s3://my-state-bucket/dev/terraform.tfstate
s3://my-state-bucket/staging/terraform.tfstate
s3://my-state-bucket/prod/terraform.tfstate
```

Workspace-Based Organization:

- Use OpenTofu workspaces for similar environments
- Separate buckets for truly different systems

State File Naming:

- Use descriptive key names
- Include environment information
- Use consistent structure

❌ Anti-Patterns

Don't:

- Store state in version control
- Share state files via email or chat
- Manually edit state files
- Use same state for unrelated infrastructure
- Skip state backups
- Ignore state locking

Don't store secrets directly in variables:

```hcl
# Bad
variable "db_password" {
  default = "SuperSecret123!"  # Will end up in state!
}

# Good
variable "db_password" {
  sensitive = true
  # Pass via environment variable or secrets manager
}
```

Summary

Key Takeaways:

1. **State is critical**: It's the mapping between your config and real resources
2. **Remote state for teams**: Always use remote state in team environments
3. **Enable locking**: Prevent concurrent modifications with DynamoDB (S3) or built-in locking (GCS, Azure)
4. **Import carefully**: Bring existing resources under management systematically
5. **Manipulate with caution**: Always backup before using state commands
6. **Secure your state**: Encrypt at rest, control access, audit changes
7. **Plan migrations**: Test in non-production first, always backup

LAB: Migrate to Remote S3 Backend and Manage Concurrent Applies

Lab Overview

Scenario: You're a DevOps engineer at a growing startup. Your infrastructure was initially managed by a single developer using local state. Now that the team is expanding, you need to migrate to a remote backend to enable team collaboration and prevent state conflicts.

What You'll Do:

Deploy infrastructure using local state
Create S3 bucket and DynamoDB table for remote state
Migrate from local to remote state
Test state locking with concurrent operations
Practice state manipulation commands (import, move, remove)
Handle lock conflicts and recovery scenarios

Learning Outcomes:

Configure and migrate to S3 remote backend
Implement state locking with DynamoDB
Safely manipulate state
Troubleshoot common state issues

Prerequisites

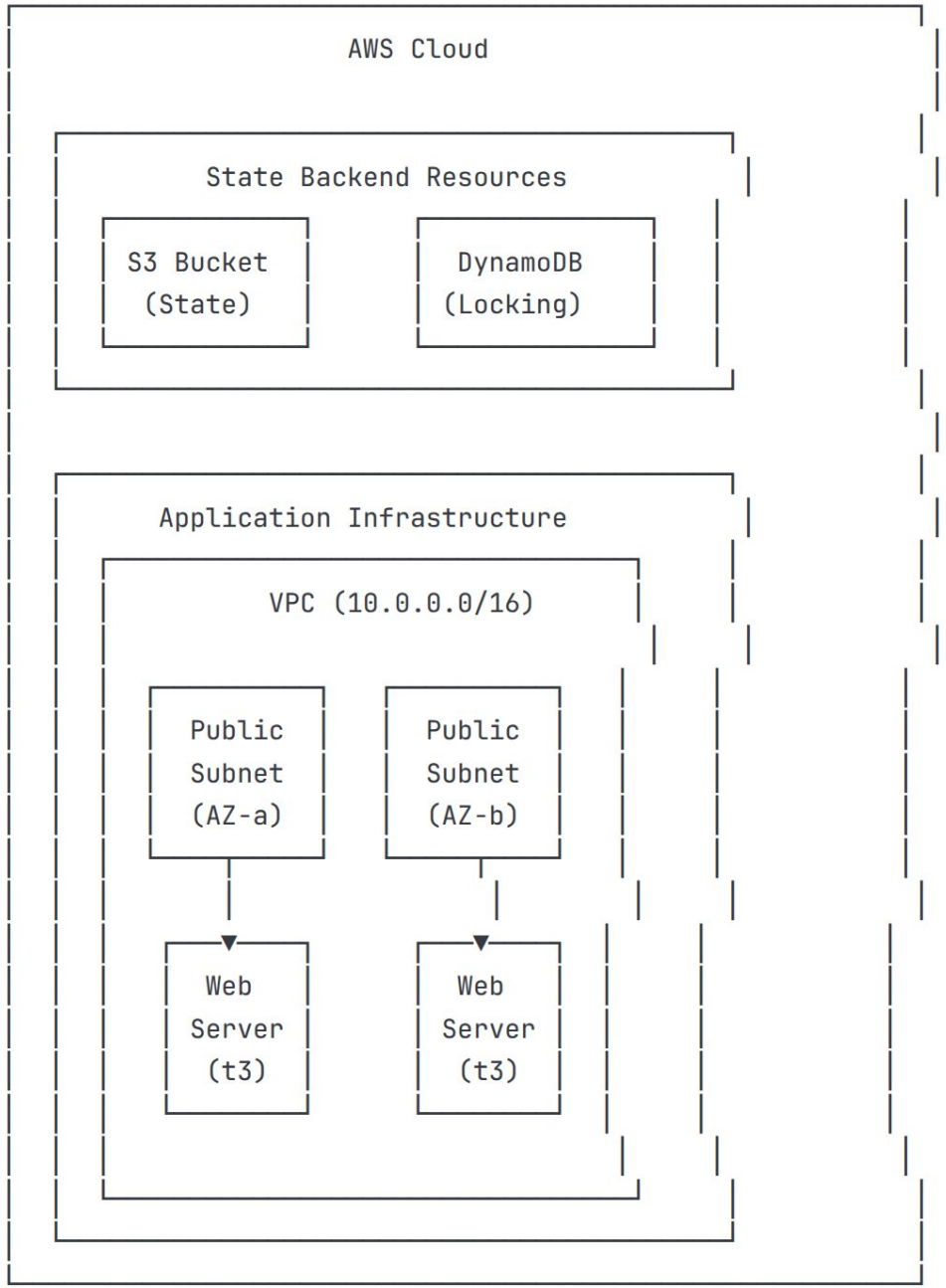
Required Tools

- OpenTofu CLI (v1.6+): `tofu --version`
- AWS CLI configured: `aws sts get-caller-identity`
- Text editor
- Terminal with multiple tabs/windows (for concurrent testing)

AWS Permissions Required

- S3: CreateBucket, PutObject, GetObject, ListBucket
- DynamoDB: CreateTable, PutItem, GetItem, DeleteItem
- EC2: CreateVpc, CreateSubnet, RunInstances, DescribeInstances
- IAM: PassRole (if using instance profiles)

Lab Architecture



OpenTofu Provisioners & Connections

Module Overview

Learning Objectives:

- Understand what provisioners are and when to use them
- Configure local-exec, remote-exec, and file provisioners
- Set up SSH connections for remote execution
- Implement error handling strategies
- Deploy and configure EC2 instances post-deployment

Prerequisites:

- Basic OpenTofu knowledge (providers, resources, variables)
- Familiarity with AWS EC2
- Basic Linux command-line skills
- Understanding of SSH concepts

1. Understanding Provisioners

What Are Provisioners?

Provisioners are OpenTofu's mechanism for executing actions on local or remote machines as part of resource creation or destruction. They allow you to:

- Run configuration management scripts
- Initialize software on newly created instances
- Execute cleanup tasks during resource destruction
- Transfer files to remote systems
- Trigger external processes

Provisioners are like "after-market scripts."

They are commands that execute either on your local machine or on the newly created server immediately after the infrastructure is built.

1. The Analogy: Building a House

- **OpenTofu (The Builder):** Builds the house. It puts up the walls, installs the roof, and connects the electricity. The resource is "created."
- **Provisioner (The Movers):** Once the house is finished, the movers come in and arrange the furniture, hang paintings, and stock the fridge.
    - *Without provisioners,* you have an empty house (a blank server).
    - *With provisioners,* you have a house ready to live in (a server with software installed).

When to Use Provisioners

⚠️ **Important:** Provisioners are a "last resort" in OpenTofu. They should be used when no better alternative exists.

Good Use Cases:

- Bootstrapping initial configuration management (Ansible, Chef, Puppet)
- One-time setup tasks that can't be handled by cloud-init
- Triggering external systems that lack API integration
- Development and testing scenarios

Better Alternatives:

- **cloud-init / user_data** - For instance initialization
- **Configuration management tools** - For ongoing configuration
- **Custom AMIs** - For pre-configured images
- **Container orchestration** - For application deployment
- **AWS Systems Manager** - For instance configuration

Provisioner Lifecycle

Provisioners run:

- **Creation-time:** After resource is created (default)
- **Destruction-time:** Before resource is destroyed (when = destroy)

Critical Limitation: Provisioners only run during resource creation or destruction, NOT during updates. This makes them unsuitable for ongoing configuration management.

4. When Should You Use Them? (The "Last Resort" Rule)

In modern DevOps, **you should avoid Provisioners if possible.**

Why?

- **They are "Dumb":** OpenTofu doesn't know what happens inside the script. If the script fails halfway, Tofu might think the server is broken ("tainted") and try to delete it next time.
- **No State:** Tofu tracks the server, but it doesn't track the *software* inside. If you change the script, Tofu won't know to update the server.

2. Types of Provisioners

2.1 local-exec Provisioner

Executes commands on the **machine running OpenTofu** (your workstation or CI/CD server).

Syntax:

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

  # Optional parameters
  working_dir = "/tmp"
  interpreter = ["/bin/bash", "-c"]
  environment = {
    FOO = "bar"
  }
}
```

Use Cases:

- Recording information locally (IP addresses, DNS names)
- Triggering local scripts or tools
- Updating local configuration files
- Calling external APIs or webhooks
- Running local validation checks

```
# Update local DNS records
provisioner "local-exec" {
  command = "aws route53 change-resource-record-sets --cli-input-json file://dns-update.json"
}

# Save instance information
provisioner "local-exec" {
  command = "echo 'Instance ${self.id} created at ${self.public_ip}' >> deployment.log"
}

# Trigger external monitoring
provisioner "local-exec" {
  command = "curl -X POST https://monitoring.example.com/register -d 'host=${self.private_ip}'"
}
```

Interpreter Options:

- Default: `["/bin/sh", "-c"]` on Unix, `["cmd", "/C"]` on Windows
- Can specify custom: `interpreter = ["/bin/bash", "-c"]`
- PowerShell: `interpreter = ["PowerShell", "-Command"]`

2.2 remote-exec Provisioner

Executes commands on the **remote resource** after it's created.

Syntax:

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

  # Or use a script file
  script = "scripts/setup.sh"

  # Or use multiple scripts
  scripts = ["scripts/install.sh", "scripts/configure.sh"]
}
```

Three Execution Methods:

1. **inline** - List of commands (most common)

```hcl
inline = [
  "sudo yum update -y",
  "sudo yum install -y docker",
  "sudo systemctl enable docker",
  "sudo systemctl start docker"
]
```

2. **script** - Single local script file to execute

```hcl
script = "${path.module}/scripts/bootstrap.sh"
```

3. **scripts** - Multiple script files in order

```hcl
scripts = [
  "${path.module}/scripts/install-deps.sh",
  "${path.module}/scripts/configure-app.sh",
  "${path.module}/scripts/start-services.sh"
]
```

Use Cases:

- Installing software packages
- Starting services
- Running configuration scripts
- Waiting for services to be ready
- Executing deployment commands

2.3 file Provisioner

Copies files or directories from local machine to remote resource.

Syntax:

```hcl
provisioner "file" {
  source      = "conf/myapp.conf"
  destination = "/etc/myapp.conf"
}

# Copy entire directory
provisioner "file" {
  source      = "configs/"  # Trailing slash important!
  destination = "/etc/myapp"
}

# Use content instead of file
provisioner "file" {
  content     = templatefile("${path.module}/config.tpl", {
    server_port = var.port
  })
  destination = "/tmp/config.txt"
}
```

Parameters:

- **source** - Local file or directory path
- **destination** - Remote path (must be writable by connection user)
- **content** - Direct content (alternative to source)

Directory Behavior:

- `source = "foo"` → Copies directory `foo` into destination
- `source = "foo/"` → Copies contents of `foo` into destination (note trailing slash)

Use Cases:

- Deploying configuration files
- Transferring application code
- Uploading SSL certificates
- Copying installation packages
- Deploying scripts before execution

Typical Pattern:

```hcl
# First, copy the script
provisioner "file" {
  source      = "${path.module}/scripts/setup.sh"
  destination = "/tmp/setup.sh"
}

# Then, execute it
provisioner "remote-exec" {
  inline = [
    "chmod +x /tmp/setup.sh",
    "sudo /tmp/setup.sh"
  ]
}
```

3. Connection Blocks

Connection blocks define how OpenTofu connects to remote resources for `remote-exec` and `file` provisioners.

SSH Connections (Linux/Unix)

hcl

```hcl
connection {
  type        = "ssh"
  host        = self.public_ip
  user        = "ec2-user"
  private_key = file("~/.ssh/id_rsa")
  timeout     = "5m"
}
```

SSH Connection Parameters:

Parameter	Description	Default	Example
`type`	Connection type	"ssh"	"ssh"
`host`	Remote host	-	`self.public_ip`
`port`	SSH port	22	2222
`user`	SSH username	"root"	"ec2-user"
`password`	SSH password	-	"secret123"
`private_key`	SSH private key content	-	`file("~/.ssh/key.pem")`
`certificate`	SSH certificate	-	`file("cert.pub")`
`agent`	Use SSH agent	true	false
`agent_identity`	Specific key from agent	-	"key-name"
`host_key`	Expected host public key	-	-
`timeout`	Connection timeout	"5m"	"10m"
`script_path`	Remote path for scripts	-	"/tmp/script.sh"

WinRM Connections (Windows)

```hcl
connection {
  type     = "winrm"
  host     = self.public_ip
  user     = "Administrator"
  password = var.admin_password
  https    = true
  insecure = true
  timeout  = "10m"
}
```

WinRM Connection Parameters:

Parameter	Description	Default
`type`	Connection type	"winrm"
`host`	Remote host	-
`port`	WinRM port	5985 (HTTP), 5986 (HTTPS)
`user`	Username	-
`password`	Password	-
`https`	Use HTTPS	false
`insecure`	Skip certificate validation	false
`use_ntlm`	Use NTLM auth	false
`timeout`	Connection timeout	"5m"

Connection Block Placement

Option 1: Inside Provisioner (Specific)

hcl

```hcl
provisioner "remote-exec" {
  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ubuntu"
    private_key = file(var.private_key_path)
  }

  inline = ["echo 'Connected!'"]
}
```

Option 2: Inside Resource (Shared by all provisioners)

hcl

```hcl
resource "aws_instance" "web" {
  # ... instance configuration ...

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ec2-user"
    private_key = file(var.private_key_path)
  }

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

  provisioner "remote-exec" {
    inline = ["sudo mv /tmp/app.conf /etc/app.conf"]
  }
}
```

Using self Object

Inside provisioners, use `self` to reference the parent resource's attributes:

hcl

```hcl
connection {
  host = self.public_ip        # Instance's public IP
  user = self.tags["SSHUser"]  # Tag value from instance
}

provisioner "local-exec" {
  command = "echo ${self.id} >> instances.txt"  # Instance ID
}
```

4. Error Handling

on_failure Behavior

Controls what happens when a provisioner fails.

Syntax:

```hcl
provisioner "remote-exec" {
  inline = ["exit 1"]  # This will fail

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

Options:

Value	Behavior
`fail`	Stop and mark resource as tainted (default)
`continue`	Log error but continue, resource marked as created

Example Use Cases:

hcl

```hcl
# Critical provisioner - must succeed
provisioner "remote-exec" {
  inline = [
    "sudo systemctl start critical-service"
  ]
  on_failure = fail  # Default behavior
}

# Optional provisioner - failure OK
provisioner "remote-exec" {
  inline = [
    "sudo apt-get update",
    "sudo apt-get upgrade -y"  # Might fail, but OK
  ]
  on_failure = continue
}

# Cleanup task - don't fail if already clean
provisioner "remote-exec" {
  inline = [
    "rm -f /tmp/setup.sh || true"
  ]
  on_failure = continue
}
```

when Parameter

Controls when provisioner runs.

Syntax:

```hcl
provisioner "local-exec" {
  when    = destroy
  command = "echo 'Instance ${self.id} being destroyed' >> cleanup.log"
}
```

Options:

Value	Runs
`create`	During resource creation (default)
`destroy`	During resource destruction

Destruction-time Example:

```hcl
resource "aws_instance" "app" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.micro"

  # Runs during creation
  provisioner "remote-exec" {
    inline = [
      "sudo systemctl start myapp"
    ]
  }

  # Runs during destruction
  provisioner "remote-exec" {
    when = destroy
    inline = [
      "sudo systemctl stop myapp",
      "sudo /opt/cleanup.sh"
    ]
  }

  # Local cleanup during destruction
  provisioner "local-exec" {
    when    = destroy
    command = "curl -X DELETE https://api.example.com/servers/${self.id}"
  }
}
```

Important Notes:

- Destroy provisioners run **before** the resource is destroyed
- If destroy provisioner fails, resource is NOT destroyed
- `self` references still work in destroy provisioners
- Connection must still be valid (resource must exist and be accessible)

Summary

Key Takeaways:

1. Provisioners are a last resort - prefer cloud-init, AMIs, or config management
2. Three types: local-exec (local), remote-exec (remote), file (transfer)
3. Connection blocks define SSH/WinRM access for remote provisioners
4. Use `on_failure` and `when` for error handling
5. Only run on create/destroy, not on updates
6. Keep provisioners simple and idempotent

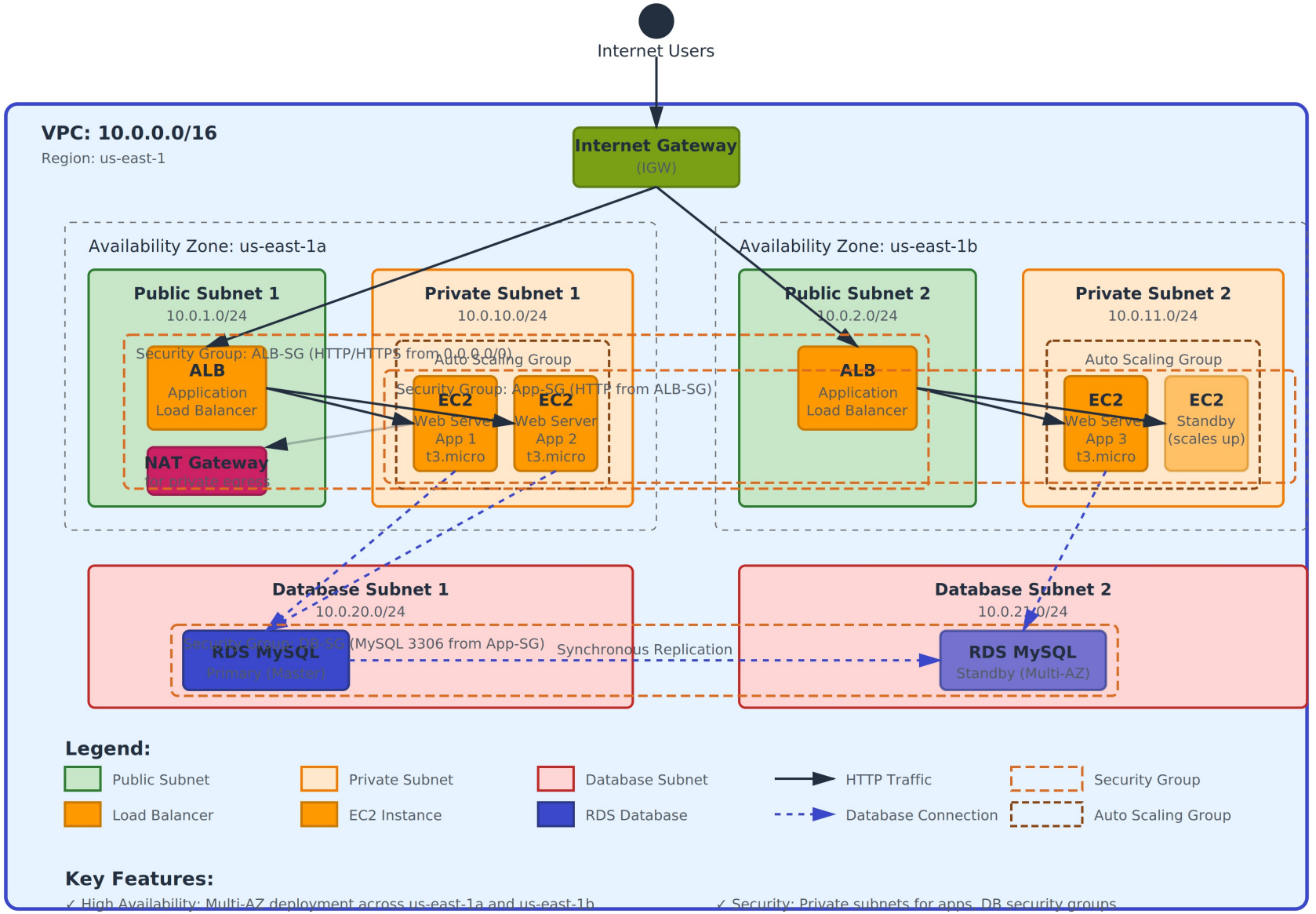
Lab 15 - Provisioners

Lab - Complex Infrastructure with Dependencies , Autoscaling, ALB

lesson6_complex_infra_w_depe_lb

Use Case 5: 3-Tier Web Application with Auto-Scaling

AWS Architecture - Load Balancer, Auto Scaling Group, and RDS

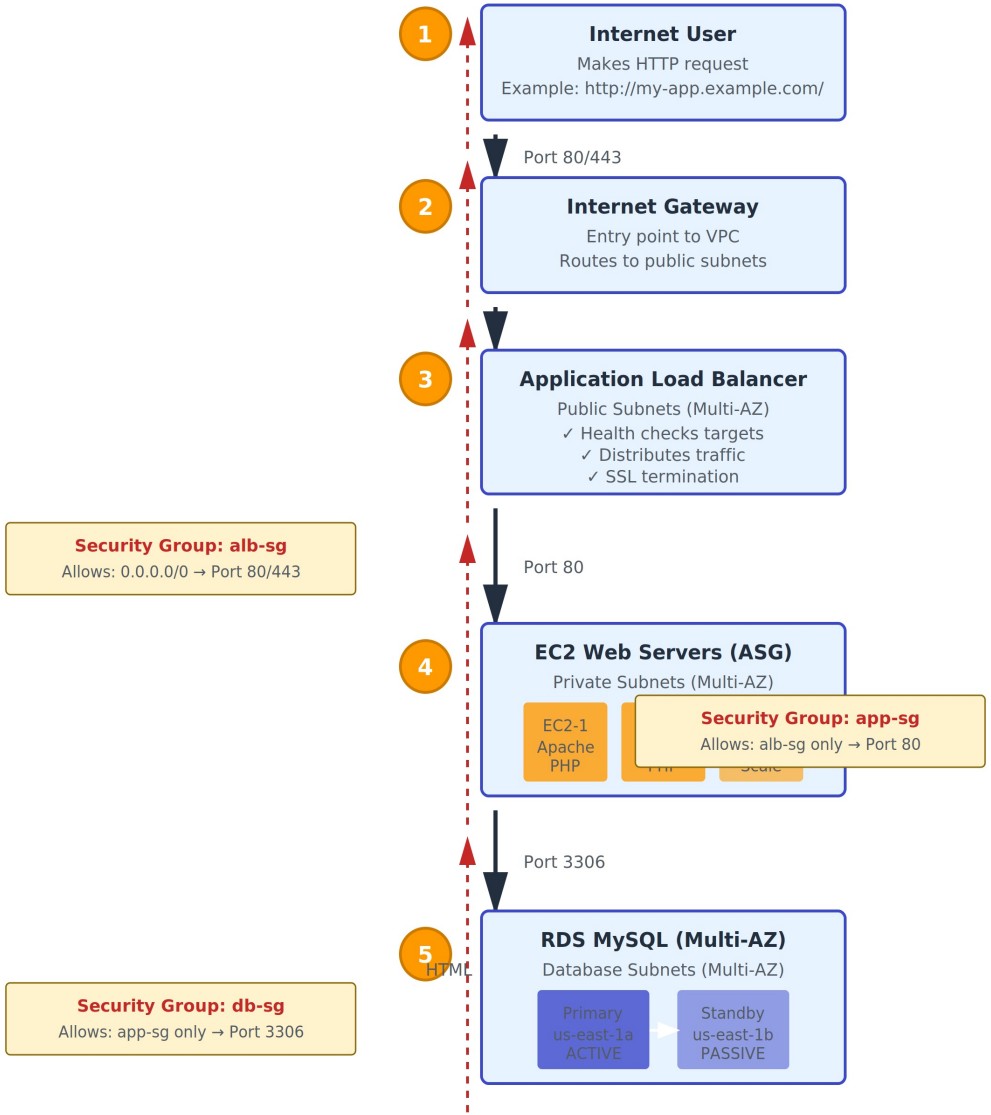


Key Features:

- ✓ High Availability: Multi-AZ deployment across us-east-1a and us-east-1b
- ✓ Auto Scaling: 2-6 EC2 instances based on CPU utilization (target: 70%)
- ✓ Load Balancing: ALB distributes traffic with health checks
- ✓ Security: Private subnets for apps, DB security groups
- ✓ Database: Multi-AZ RDS with automatic failover
- ✓ Outbound Access: NAT Gateway for private subnet internet access

Request Flow: User to Database

Complete traffic path through 3-tier architecture



Response Flow (Returns via same path)

Key Security Points:

- Each tier has its own security group
- Security groups reference each other (not IP ranges)
- Database has NO internet access
- Application servers in private subnets
- Only ALB is publicly accessible

High Availability:

- All tiers span 2 Availability Zones
- ALB distributes across healthy targets
- Auto Scaling replaces failed instances
- RDS automatic failover (Multi-AZ)
- No single point of failure

Hands-On Lab: EC2 Post-Deployment Configuration with Provisioners

Lab Overview

Objective: Deploy an EC2 instance and use provisioners to:

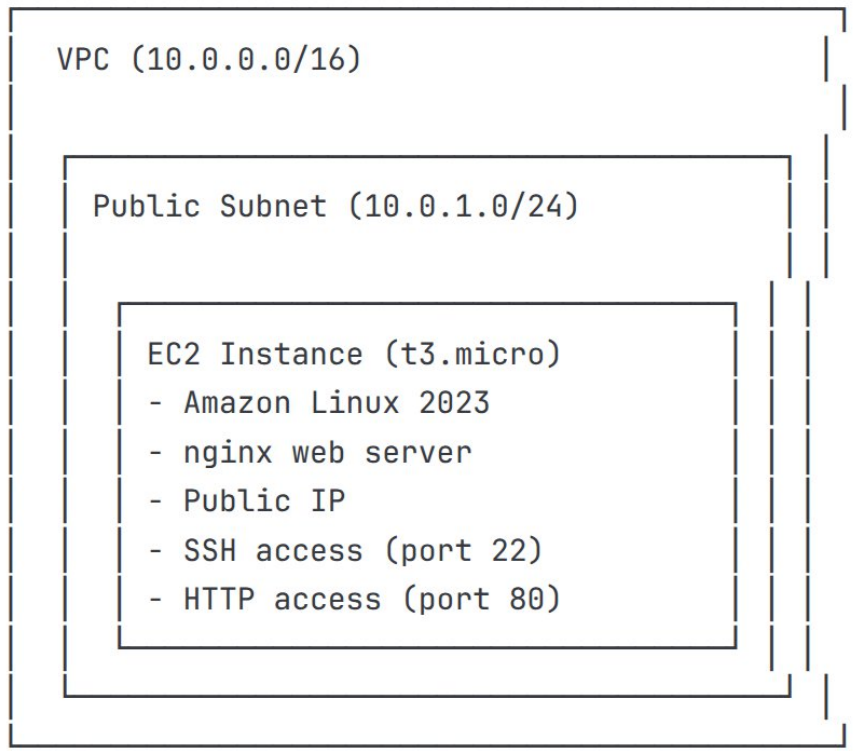
1. Install and configure a web server (nginx)
2. Deploy a custom web page
3. Configure application settings
4. Test connectivity and verify configuration

Scenario: You're deploying a web application server that needs post-deployment configuration. The application requires nginx installation, custom configuration files, and a welcome page with dynamic content.

What You'll Learn:

- Configure all three provisioner types
- Set up SSH connections
- Handle provisioner errors
- Transfer files to remote instances
- Execute remote configuration commands
- Verify provisioner success

Lab Architecture




```
Provisioner Flow:
1. file: Copy nginx config → /tmp/nginx.conf
2. file: Copy welcome page → /tmp/index.html
3. remote-exec: Install nginx, move files, start service
4. local-exec: Record deployment info locally
```

Prerequisites

Before starting this lab, ensure you have:

- [ ] AWS CLI configured with valid credentials
- [ ] OpenTofu installed (v1.6.0+)
- [ ] SSH key pair available (or script will create one)
- [ ] Permissions to create VPC, EC2, and security group resources
- [ ] Network connectivity to AWS

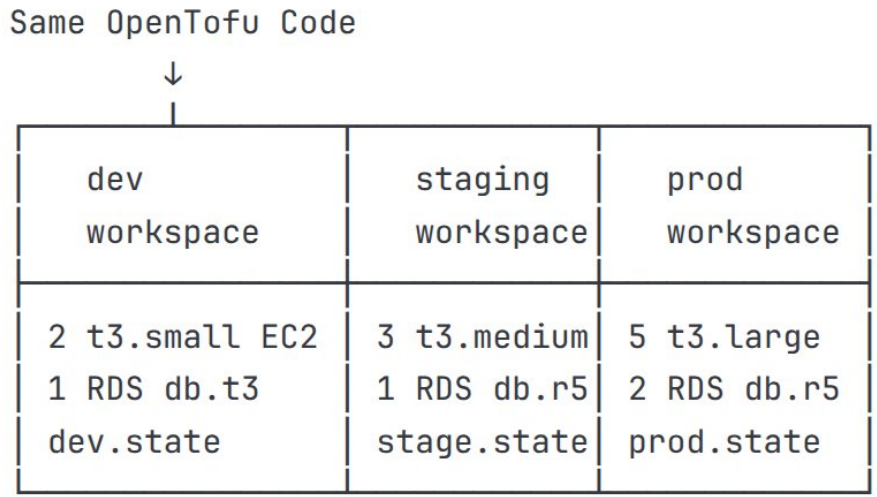
OpenTofu Workspaces - Concepts & Fundamentals

What Are Workspaces?

Workspaces in OpenTofu allow you to manage multiple distinct sets of infrastructure resources using the same configuration code. Each workspace has its own separate state file, enabling you to maintain different environments (dev, staging, production) from a single codebase.

The Core Concept

Think of workspaces as parallel universes for your infrastructure:



Each workspace maintains:

- **Independent state file**: No state pollution between environments
- **Same configuration code**: DRY principle for infrastructure
- **Isolated resources**: Resources in one workspace don't affect others
- **Separate lifecycle**: Create/update/destroy independently

How Workspaces Work Internally

Default Workspace

Every OpenTofu project starts with a `default` workspace:

```bash
# Initialize a new project
tofu init

# Check current workspace
tofu workspace show
# Output: default

# List all workspaces
tofu workspace list
# Output: * default  (asterisk shows current workspace)
```

Creating Additional Workspaces

bash

```
# Create new workspace
tofu workspace new dev
# Output: Created and switched to workspace "dev"

# Create more workspaces
tofu workspace new staging
tofu workspace new prod

# List all workspaces
tofu workspace list
# Output:
#   default
#   dev
#   staging
# * prod
```

Workspace State Storage

Local Backend

When using local state, workspaces create separate state files:

```
.terraform/
├── terraform.tfstate.d/
│   ├── dev/
│   │   └── terraform.tfstate
│   ├── staging/
│   │   └── terraform.tfstate
│   └── prod/
│       └── terraform.tfstate
└── terraform.tfstate  # default workspace state
```

Remote Backend (S3)

With S3 backend, workspaces use key prefixes:

```
s3://my-terraform-state/
├── env:/dev/myapp/terraform.tfstate
├── env:/staging/myapp/terraform.tfstate
├── env:/prod/myapp/terraform.tfstate
└── myapp/terraform.tfstate  # default workspace
```

Workspace Interpolation

The most powerful feature is `terraform.workspace` interpolation:

Example 1: Environment-Specific Naming

```hcl
# main.tf
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = terraform.workspace == "prod" ? "t3.large" : "t3.small"

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

Results across workspaces:

- **dev workspace**: Creates `dev-web-server` with `t3.small`
- **staging workspace**: Creates `staging-web-server` with `t3.small`
- **prod workspace**: Creates `prod-web-server` with `t3.large`

Workspace Interpolation

Example 2: Count-Based Scaling

hcl

```hcl
# variables.tf
variable "instance_count" {
  type = map(number)
  default = {
    dev     = 1
    staging = 2
    prod    = 5
  }
}

# main.tf
resource "aws_instance" "app" {
  count         = var.instance_count[terraform.workspace]
  ami           = var.ami_id
  instance_type = "t3.small"

  tags = {
    Name = "${terraform.workspace}-app-${count.index + 1}"
  }
}
```

Workspace Interpolation

Example 3: Locals for Complex Logic

hcl

```hcl
# locals.tf
locals {
  environment_config = {
    dev = {
      instance_type = "t3.small"
      instance_count = 1
      enable_monitoring = false
      backup_retention = 1
    }
    staging = {
      instance_type = "t3.medium"
      instance_count = 2
      enable_monitoring = true
      backup_retention = 7
    }
    prod = {
      instance_type = "t3.large"
      instance_count = 5
      enable_monitoring = true
      backup_retention = 30
    }
  }

  # Get config for current workspace
  env_config = local.environment_config[terraform.workspace]
}

# main.tf
resource "aws_instance" "app" {
  count         = local.env_config.instance_count
  ami           = var.ami_id
  instance_type = local.env_config.instance_type

  monitoring = local.env_config.enable_monitoring

  tags = {
    Name        = "${terraform.workspace}-app-${count.index + 1}"
    Environment = terraform.workspace
  }
}

resource "aws_db_instance" "main" {
  identifier           = "${terraform.workspace}-database"
  engine               = "postgres"
  instance_class       = terraform.workspace == "prod" ? "db.r5.large" : "db.t3.micro"
  backup_retention_period = local.env_config.backup_retention

  tags = {
    Environment = terraform.workspace
  }
}
```

Workspace Commands Reference

Basic Operations

bash

```
# Create and switch to new workspace
tofu workspace new <name>

# Switch to existing workspace
tofu workspace select <name>

# List all workspaces
tofu workspace list

# Show current workspace
tofu workspace show

# Delete a workspace (must be empty)
tofu workspace delete <name>
```

Common Workflows

bash

```
# Deploy to dev environment
tofu workspace select dev
tofu plan
tofu apply

# Switch to staging and deploy
tofu workspace select staging
tofu apply

# Check production state without switching
tofu workspace select prod
tofu state list
tofu workspace select dev  # switch back
```

When to Use Workspaces

✅ Good Use Cases

1. **Similar Infrastructure, Different Scale**
    - Same VPC layout, different CIDR blocks
    - Same application tier, different instance counts
    - Same architecture, different sizing
2. **Development Workflow**
    - Feature branches that need temporary environments
    - Developer sandboxes from same template
    - Testing infrastructure changes
3. **Single-Team Management**
    - One team manages all environments
    - Consistent access patterns
    - Shared responsibility model
4. **Rapid Environment Provisioning**
    - Spin up identical environments quickly
    - Testing disaster recovery procedures
    - Demo environments

Workspace vs Other Patterns

Workspaces:

```
my-app/
├── main.tf
├── variables.tf
└── outputs.tf

# One codebase, multiple workspaces
```

Separate Directories:

```
environments/
├── dev/
│   ├── main.tf
│   └── variables.tf
├── staging/
│   ├── main.tf
│   └── variables.tf
└── prod/
    ├── main.tf
    └── variables.tf
```

When to use each:

- **Workspaces**: Infrastructure is 90%+ identical
- **Directories**: Significant differences between environments

Workspace vs Separate Repositories

Workspaces:

```
infrastructure-repo/
└── app-platform/
    ├── main.tf
    └── ...
```

Separate Repositories:

```
app-platform-dev/
app-platform-staging/
app-platform-prod/
```

When to use each:

- **Workspaces**: Same team, similar infrastructure
- **Repositories**: Different teams, different compliance requirements

Common Patterns

Pattern 1: Variable Files per Workspace

bash

```
# Directory structure
.
├── main.tf
├── variables.tf
├── dev.tfvars
├── staging.tfvars
└── prod.tfvars
```

bash

```
# Usage
tofu workspace select dev
tofu apply -var-file="dev.tfvars"

tofu workspace select prod
tofu apply -var-file="prod.tfvars"
```

Common Patterns

Pattern 2: Workspace Validation

hcl

```hcl
# main.tf
locals {
  valid_workspaces = ["dev", "staging", "prod"]

  # Validate workspace name
  validate_workspace = contains(local.valid_workspaces, terraform.workspace)
    ? true
    : file("ERROR: Invalid workspace. Must be one of: ${join(", ", local.valid_workspaces)}")
}
```

Common Patterns

Pattern 3: Conditional Resources

hcl

```hcl
# Only create in production
resource "aws_cloudwatch_dashboard" "main" {
  count          = terraform.workspace == "prod" ? 1 : 0
  dashboard_name = "production-dashboard"
  # ... configuration
}

# Only create in non-production
resource "aws_instance" "debug" {
  count = terraform.workspace != "prod" ? 1 : 0
  # ... debug instance configuration
}
```

Pattern 4: Workspace-Specific Backend Config

hcl

```hcl
# backend.tf
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
    key            = "app/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"

    # Workspace prefix creates: env:/dev/, env:/staging/, env:/prod/
    workspace_key_prefix = "env"
  }
}
```

Best Practices

1. Always Verify Current Workspace

bash

```bash
# Before applying changes
tofu workspace show
tofu plan | grep "workspace"
```

2. Use Workspace Naming Conventions

bash

```bash
# Good
dev, staging, prod

# Also good
dev-useast1, dev-uswest2
feature-auth, feature-payments

# Bad
johns-test, temporary, new, old
```

3. Document Workspace Strategy

markdown

```markdown
# README.md
## Workspaces
- `dev`: Development environment (auto-deploys from main)
- `staging`: Pre-production testing (manual deploys)
- `prod`: Production (requires approval)
```

4. Implement Safety Checks

bash

```bash
# Add to CI/CD pipeline
if [ "$(tofu workspace show)" == "prod" ]; then
  echo "Deploying to PRODUCTION. Continue? (yes/no)"
  read confirmation
  [ "$confirmation" != "yes" ] && exit 1
fi
```

Key Limitations

1. **No Cross-Workspace References**
    - Can't reference resources from other workspaces
    - Each workspace is completely isolated
2. **Workspace State is Mutable**
    - Deleting workspace deletes all resource state
    - No automatic protection for production workspaces
3. **Limited Backend Support**
    - Not all backends support workspaces
    - Some cloud providers have better native solutions
4. **No Built-in RBAC**
    - Everyone with state access can use any workspace
    - Need external controls for production protection

Summary

Workspaces are a powerful feature for managing multiple environments from a single codebase, but they're not a one-size-fits-all solution. Use them when:

- Infrastructure is fundamentally similar across environments
- You need rapid environment creation
- Single team manages all environments
- State isolation is sufficient for your security needs

Consider alternatives when:

- Environments have significantly different architectures
- Different teams need different access controls
- Regulatory requirements demand hard separation
- Environments have different lifecycles

Environment Management with Workspaces

Overview

This section covers practical patterns for managing dev, staging, and production environments using OpenTofu workspaces. We'll explore configuration strategies, variable management, and real-world deployment patterns.

Environment Design Philosophy

The Three-Tier Standard

Most organizations use a three-tier environment structure:

```
Development (dev)
    ↓
Staging (staging/stage)
    ↓
Production (prod)
```

Environment Characteristics:

Aspect	Dev	Staging	Production
Purpose	Active development	Pre-prod testing	Live users
Size	Small (cost-optimized)	Medium (prod-like)	Full scale
Data	Synthetic/scrambled	Sanitized prod copy	Real user data
Uptime	Best effort	High	Critical
Changes	Multiple per day	Daily/weekly	Controlled releases
Cost	Minimize	Balance	Optimize for performance

Workspace Mapping

hcl

```
# Workspace to environment mapping
dev      → Development environment
staging  → Staging/QA environment
prod     → Production environment
```

Configuration Strategies

Strategy 1: Inline Conditionals

Simple conditional logic directly in resources:

```hcl
# main.tf
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = terraform.workspace == "prod" ? "t3.large" : "t3.small"

  monitoring = terraform.workspace == "prod" ? true : false

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

Pros:

- Simple and direct
- Easy to understand
- No extra files

Cons:

- Can become complex quickly
- Hard to maintain with many differences
- Limited to simple conditionals

Strategy 2: Variable Maps

Define environment-specific values in variable maps:

hcl

```hcl
# variables.tf
variable "instance_config" {
  type = map(object({
    instance_type  = string
    instance_count = number
    volume_size    = number
  }))

  default = {
    dev = {
      instance_type  = "t3.small"
      instance_count = 1
      volume_size    = 20
    }
    staging = {
      instance_type  = "t3.medium"
      instance_count = 2
      volume_size    = 50
    }
    prod = {
      instance_type  = "t3.large"
      instance_count = 5
      volume_size    = 100
    }
  }
}
```

```hcl
# main.tf
resource "aws_instance" "app" {
  count         = var.instance_config[terraform.workspace].instance_count
  ami           = var.ami_id
  instance_type = var.instance_config[terraform.workspace].instance_type

  root_block_device {
    volume_size = var.instance_config[terraform.workspace].volume_size
  }

  tags = {
    Name = "${terraform.workspace}-app-${count.index + 1}"
  }
}
```

Pros:

- Clear separation of configuration
- Easy to add new environments
- Type-safe with object validation

Cons:

- All values in version control
- Can't override easily per deployment
- Variables file can get large

Strategy 3: Separate Variable Files

External variable files for each environment:

hcl

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

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

variable "enable_monitoring" {
  type        = bool
  description = "Enable detailed CloudWatch monitoring"
}
```

hcl

```hcl
# dev.tfvars
instance_type     = "t3.small"
instance_count    = 1
enable_monitoring = false
```

hcl

```hcl
# staging.tfvars
instance_type     = "t3.medium"
instance_count    = 2
enable_monitoring = true
```

hcl

```hcl
# prod.tfvars
instance_type     = "t3.large"
instance_count    = 5
enable_monitoring = true
```

bash

```bash
# Usage
tofu workspace select dev
tofu apply -var-file="dev.tfvars"
```

Pros:

- Clean separation of concerns
- Easy to override values
- Can use different sources (git, vault, etc.)

Cons:

- Need to remember correct var file
- Easy to apply wrong file to wrong workspace
- Extra files to manage

Strategy 4: Locals with Complex Logic (Recommended)

Combine workspace interpolation with locals for complex configurations:

hcl

```hcl
# locals.tf
locals {
  # Environment-specific configurations
  environments = {
    dev = {
      # Compute
      instance_type      = "t3.small"
      instance_count     = 1
      min_size           = 1
      max_size           = 2

      # Database
      db_instance_class  = "db.t3.micro"
      db_allocated_storage = 20
      db_multi_az        = false

      # Networking
      vpc_cidr           = "10.0.0.0/16"

      # Features
      enable_monitoring  = false
      enable_backup      = false
      backup_retention   = 1

      # Scaling
      enable_autoscaling = false
    }

    staging = {
      # Compute
      instance_type      = "t3.medium"
      instance_count     = 2
      min_size           = 2
      max_size           = 4

      # Database
      db_instance_class  = "db.t3.small"
      db_allocated_storage = 50
      db_multi_az        = false

      # Networking
      vpc_cidr           = "10.1.0.0/16"

      # Features
      enable_monitoring  = true
      enable_backup      = true
      backup_retention   = 7

      # Scaling
      enable_autoscaling = true
    }

    prod = {
      # Compute
      instance_type      = "t3.large"
      instance_count     = 5
      min_size           = 3
      max_size           = 10

      # Database
      db_instance_class  = "db.r5.large"
      db_allocated_storage = 100
      db_multi_az        = true

      # Networking
      vpc_cidr           = "10.2.0.0/16"

      # Features
      enable_monitoring  = true
      enable_backup      = true
      backup_retention   = 30

      # Scaling
      enable_autoscaling = true
    }
  }
}
```

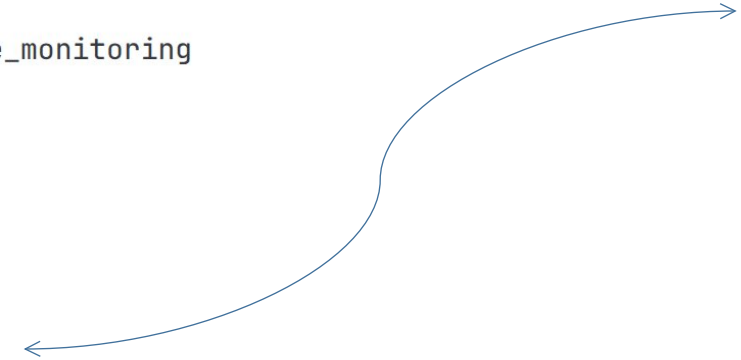
Strategy 4: Locals with Complex Logic (Recommended)

```
  # Get config for current workspace
  env = local.environments[terraform.workspace]

  # Common tags applied to all resources
  common_tags = {
    Environment = terraform.workspace
    ManagedBy   = "OpenTofu"
    Project     = "webapp"
    CostCenter  = terraform.workspace == "prod" ? "productio : "non-production"
  }
}
```

Strategy 4: Locals with Complex Logic (Recommended)

```hcl
# main.tf - Clean resource definitions
resource "aws_instance" "app" {
  count         = local.env.instance_count
  ami           = var.ami_id
  instance_type = local.env.instance_type

  monitoring = local.env.enable_monitoring

  root_block_device {
    volume_size = 30
  }

  tags = merge(
    local.common_tags,
    {
      Name = "${terraform.workspace}-app-${count.index + 1}"
    }
  )
}
```

```hcl
resource "aws_db_instance" "main" {
  identifier              = "${terraform.workspace}-database
  engine                  = "postgres"
  engine_version          = "15.3"
  instance_class          = local.env.db_instance_class
  allocated_storage       = local.env.db_allocated_storage

  multi_az                = local.env.db_multi_az
  backup_retention_period = local.env.backup_retention

  skip_final_snapshot = terraform.workspace != "prod"

  tags = local.common_tags
}
```



[Strategy 4 Combine workspace]

Pros:

- Clean separation of config and infrastructure code
- Type-safe and validated
- Easy to see all environment differences
- Supports complex logic
- Single source of truth

Cons:

- All values in code (use variables for secrets)
- Requires workspace discipline

Hands-On Labs: Workspaces & Environment Segregation

Lab Overview

This hands-on lab series will take you through implementing workspaces for environment segregation.

We'll deploy real AWS infrastructure across dev, staging, and production environments.

Lab Structure:
Lab : Basic workspace usage with simple web servers

Prerequisites:

- AWS account with admin access
- AWS CLI configured
- OpenTofu >= 1.6.0 installed
- SSH key pair available
- Basic understanding of AWS (EC2, VPC, S3)

Scenario

You'll deploy a simple web application across three environments (dev, staging, prod) with different instance counts and sizes per environment.

Architecture

```
Development:     1x t3.small  + VPC
Staging:         2x t3.medium + VPC
Production:      3x t3.large  + VPC + Monitoring
```

OPA Fundamentals - Understanding Policy-as-Code

1. Introduction to Policy-as-Code

What is Policy-as-Code?

Policy-as-Code is the practice of writing organizational policies, compliance rules, and security standards as executable code rather than documentation. Just as Infrastructure-as-Code (IaC) transformed infrastructure management, Policy-as-Code transforms governance.

Why Policy-as-Code Matters

Why Policy-as-Code Matters

Traditional Approach Problems:

- Manual reviews are slow and error-prone
- Policies documented in wikis get outdated
- Inconsistent enforcement across teams
- No way to test policies before applying them
- Difficult to audit compliance

Why Policy-as-Code Matters

Policy-as-Code Benefits:

- Automated, consistent enforcement
- Policies version controlled alongside infrastructure
- Testable before deployment
- Fast feedback to developers
- Clear audit trail
- Shift-left security (catch issues early)

Why Policy-as-Code Matters

Real-World Scenario

Imagine you're managing infrastructure for a healthcare company:

- HIPAA requires encryption at rest and in transit
- Finance requires cost controls on resources
- Security requires specific tagging for compliance tracking
- Architecture requires resources in approved regions only

Without Policy-as-Code:

- Manual review of every pull request (slow)
- Mistakes slip through (risky)
- Policies drift between teams (inconsistent)
- Violations discovered after deployment (expensive)

Why Policy-as-Code Matters

With Policy-as-Code:

- Automatic validation on every commit
- Instant feedback to developers
- Consistent enforcement across all teams
- Issues caught before any resources are created

2. Open Policy Agent (OPA) Architecture

What is OPA?

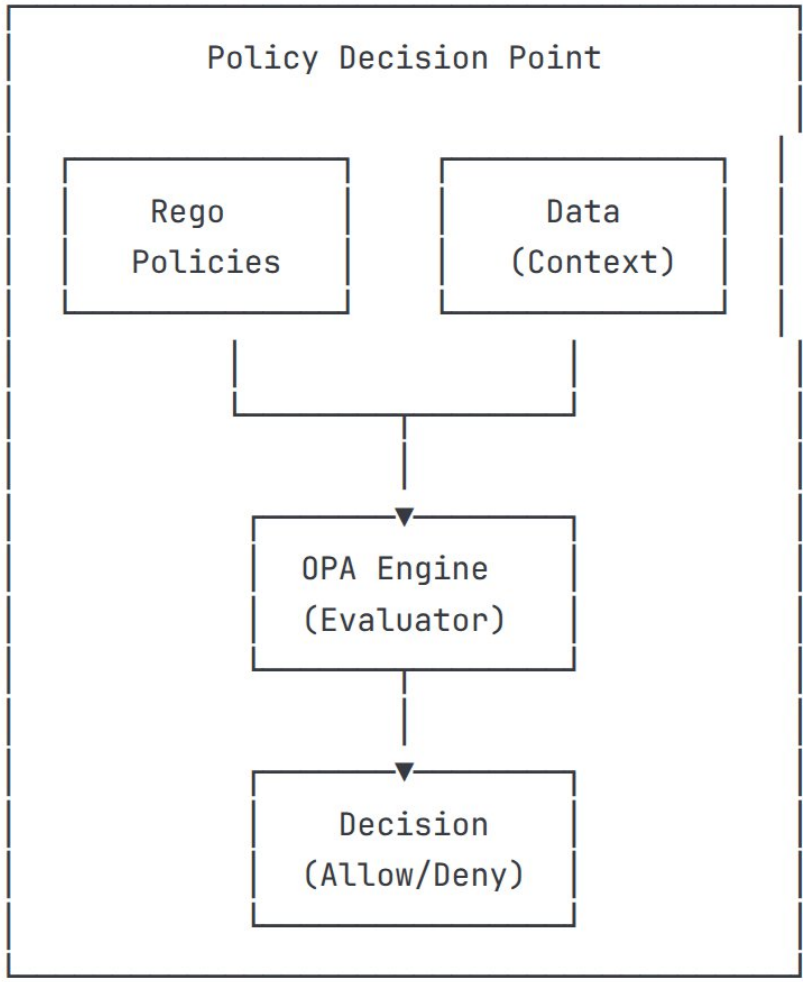
Open Policy Agent is a general-purpose policy engine that:

- Evaluates policies written in the Rego language
- Works with any JSON/YAML data
- Provides a unified way to enforce policies
- Can be embedded in applications or used as a service

Key Components:

1. **Rego Policies:** Rules written in the Rego language
2. **Data/Context:** Information about what's being evaluated
3. **OPA Engine:** Evaluates policies against data
4. **Decision:** Result (allow/deny) with reasoning

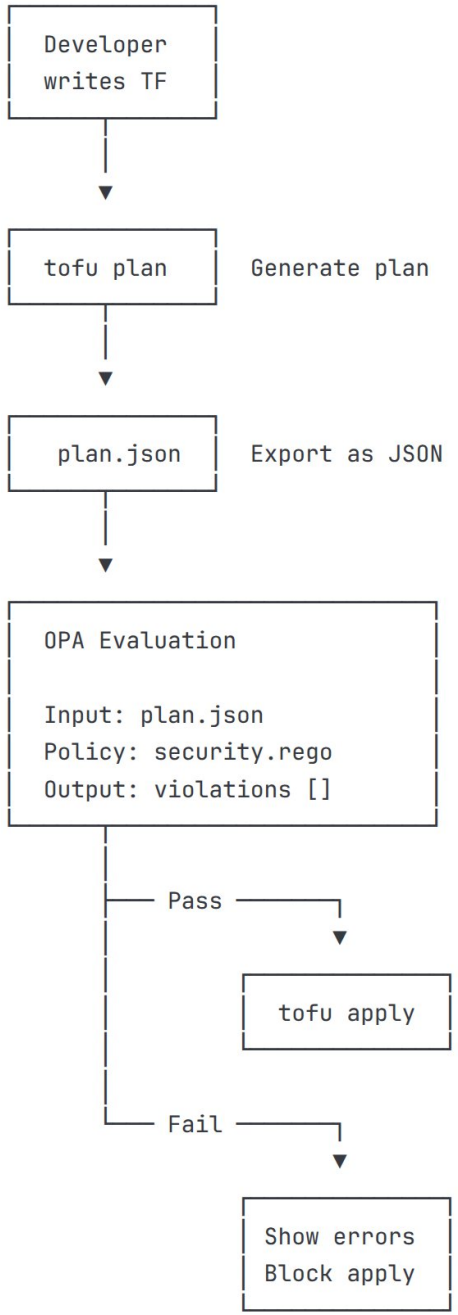
OPA Components



How OPA Works with OpenTofu

Policy Evaluation Workflow

1. **Generate Plan:** OpenTofu creates execution plan
2. **Export JSON:** Convert plan to JSON format
3. **Load Policy:** OPA loads relevant Rego policies
4. **Evaluate:** OPA checks plan against policies
5. **Report:** OPA returns violations or approval
6. **Decide:** Allow apply or require fixes



3. OPA vs Other Policy Tools

Comparison Matrix

Feature	OPA	Sentinel	Cloud Custodian
Language	Rego	HCL-like	YAML
Cloud-agnostic	Yes	No (HashiCorp)	Partial
Runtime	Standalone	Terraform Enterprise	Cloud-specific
Testing	Built-in	Limited	Limited
Complexity	Moderate	Low	Low-Moderate
Flexibility	Very High	Moderate	Moderate

How OPA Works with OpenTofu

When to Use OPA

Best For:

- Multi-cloud environments
- Complex policy logic
- Microservices authorization
- Kubernetes admission control
- Organizations wanting open-source solutions

Consider Alternatives If:

- Using only HashiCorp Terraform Enterprise (Sentinel)
- Simple policies only (might be overkill)
- Team unfamiliar with logic programming

4. Understanding the Rego Language

What is Rego?

Rego (pronounced "ray-go") is a declarative language designed for policy writing:

- Based on Datalog (logic programming)
- Focused on querying and transforming data
- Optimized for policy evaluation
- Similar to SQL in purpose, different in syntax

How OPA Works with OpenTofu

Rego Philosophy

Declarative, Not Imperative:

```rego
# Rego: Declare what should be true
deny[msg] {
    resource := input.resource_changes[_]
    resource.type == "aws_s3_bucket"
    not resource.change.after.encryption
    msg := "S3 buckets must be encrypted"
}

# Versus imperative (pseudocode):
# for each resource in plan:
#     if resource.type == "s3_bucket":
#         if not resource.encrypted:
#             return "error"
```

Key Characteristics:

- Rules evaluate to true or false
- No variables in traditional sense (only bindings)
- No side effects or state changes
- Order-independent execution

How OPA Works with OpenTofu

5. Setting Up OPA with OpenTofu

Installation

Install OPA:

```
# Linux
curl -L -o opa https://openpolicyagent.org/downloads/latest/opa_linux_amd64
chmod +x opa
sudo mv opa /usr/local/bin/

# macOS
brew install opa

# Windows
curl -L -o opa.exe https://openpolicyagent.org/downloads/latest/opa_windows_amd64.exe

# Verify
opa version
```

Directory Structure

```
project/
├── infrastructure/
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
├── policies/
│   ├── tagging.rego
│   ├── security.rego
│   ├── cost.rego
│   └── test/
│       ├── tagging_test.rego
│       ├── security_test.rego
│       └── cost_test.rego
└── scripts/
    ├── check-policies.sh
    └── generate-plan.sh
```

Basic Workflow Script

bash

```bash
#!/bin/bash
# scripts/check-policies.sh

set -e

echo "🔄 Generating OpenTofu plan..."
cd infrastructure
tofu plan -out=tfplan.binary

echo "📄 Converting plan to JSON..."
tofu show -json tfplan.binary > tfplan.json

echo "🔍 Evaluating policies..."
cd ../policies
opa eval --data . --input ../infrastructure/tfplan.json \
    --format pretty \
    "data.terraform.deny"

if [ $? -eq 0 ]; then
    echo "✅ All policies passed!"
    exit 0
else
    echo "❌ Policy violations found!"
    exit 1
fi
```

6. OpenTofu Plan Structure

Understanding the JSON plan structure is crucial for writing policies.

Plan JSON Overview

```json
{
  "format_version": "1.2",
  "terraform_version": "1.6.0",
  "planned_values": {
    "root_module": {
      "resources": [...]
    }
  },
  "resource_changes": [
    {
      "address": "aws_instance.web",
      "mode": "managed",
      "type": "aws_instance",
      "name": "web",
      "provider_name": "registry.opentofu.org/hashicorp/aws",
      "change": {
        "actions": ["create"],
        "before": null,
        "after": {
          "ami": "ami-12345678",
          "instance_type": "t3.micro",
          "tags": {
            "Name": "web-server"
          }
        },
        "after_unknown": {...}
      }
    }
  ],
  "configuration": {...}
}
```

Key Sections for Policy Writing

resource_changes: Most important for policies - contains all resource modifications:

- `actions`: ["create"], ["update"], ["delete"], ["create", "delete"] (replace)
- `before`: State before changes (null for new resources)
- `after`: State after changes (null for deleted resources)
- `type`: Resource type (aws_instance, aws_s3_bucket, etc.)

Example Access in Rego:

rego

```rego
# Get all resources being created
resource := input.resource_changes[_]
resource.change.actions[_] == "create"

# Get resource type
resource.type == "aws_s3_bucket"

# Get configuration values
resource.change.after.instance_type == "t3.large"
```

```
package aws.compliance

import rego.v1

# Helper: Define allowed types set
allowed_types := {"t2.micro", "t3.micro"}

# Rule 1: Restrict EC2 Instance Types
deny contains msg if {
    # Find all resources in the plan
    resource := input.resource_changes[_]

    # Filter for EC2 instances only
    resource.type == "aws_instance"

    # Check if the instance type is in the allowed list
    not allowed_types[resource.change.after.instance_type]

    # Return a specific error message
    msg := sprintf(
        "Invalid instance type: '%v'. Allowed types are: %v",
        [resource.change.after.instance_type, allowed_types]
    )
}

# Rule 2: Enforce Mandatory 'Environment' Tag
deny contains msg if {
    resource := input.resource_changes[_]
    resource.type == "aws_instance"

    # Check if 'Environment' tag exists
    not resource.change.after.tags.Environment

    msg := "EC2 instance is missing the mandatory 'Environment' tag."
}
```

3. How it Works (Evaluation)

If you evaluate the **Policy** against the **Input**, OPA will return two violations:

1. **Violation 1:** The input uses `t2.large`, but the policy only allows `t2.micro` or `t3.micro`.
2. **Violation 2:** The input has a `Name` tag, but is missing the required `Environment` tag.

How to test this locally

If you have the OPA binary installed, you can run this command in your terminal:

Bash

```
opa eval --data policy.rego --input input.json "data.aws.compliance.deny"
```

```
jilg@node-214:~/tofu_infra/lesson17_OPA$ opa eval --data policy.rego --input input.json "data.aws.compliance.deny"
{
  "result": [
    {
      "expressions": [
        {
          "value": [
            "EC2 instance is missing the mandatory 'Environment' tag.",
            "Invalid instance type: 't2.large'. Allowed types are: {\"t2.micro\", \"t3.micro\"}"
          ],
          "text": "data.aws.compliance.deny",
          "location": {
            "row": 1,
            "col": 1
          }
        }
      ]
    }
  ]
}
```

Rego Language Basics - Writing Effective Policies

Refer - Rego_language_tutorial.html

Hands-On Lab: OPA Policies for Security Compliance

Lab Overview

Difficulty: Intermediate

Objectives:
Write OPA policies for real-world compliance requirements
Test policies against OpenTofu plans
Fix policy violations in infrastructure code
Integrate policies into development workflow

Lab Environment Setup

Prerequisites

Tools Installed:
OpenTofu CLI
OPA CLI
AWS CLI (configured)
jq
Git

2. **Verify Installation:**

bash

```
tofu version      # Should show v1.6.0 or higher
opa version       # Should show v0.60.0 or higher
aws --version     # Should show aws-cli/2.x
jq --version      # Should show jq-1.6 or higher
```

lesson19-OPA_Policies_lab

OpenTofu + Spacelift Integration: Deploy an EC2 Instance

Integrating OpenTofu with Spacelift and deploying an AWS EC2 instance.

Overview

What you'll accomplish:

- Set up a Spacelift account and connect it to your VCS (GitHub/GitLab)
- Configure AWS credentials using Spacelift's native cloud integration
- Write OpenTofu code to provision an EC2 instance
- Create a Spacelift Stack and deploy infrastructure

Architecture Flow:

```
GitHub Repo → Spacelift Stack → AWS Cloud Integration → EC2 Instance
```

Prerequisites

1. **Spacelift Account** - Sign up at spacelift.io (14-day free trial available)
2. **AWS Account** - With permissions to create IAM roles and EC2 instances
3. **GitHub/GitLab Account** - To host your OpenTofu code
4. **Git** - To push your code to the repository

GraphQL Fundamentals

Understanding GraphQL Queries

GraphQL is a query language for APIs that allows you to request exactly the data you need:

Key Concepts:

- **Queries:** Read operations to fetch data
- **Mutations:** Write operations to modify data
- **Fields:** Specific data points you want to retrieve
- **Arguments:** Parameters to filter or customize results

Basic Structure:

graphql

```graphql
query {
  stacks {
    id
    name
    state
  }
}
```

Spacelift GraphQL Schema

Common objects you'll work with:

1. **Stack:** Infrastructure stack configuration
    - Fields: `id`, `name`, `state`, `repository`, `branch`
2. **Run:** Deployment execution
    - Fields: `id`, `state`, `type`, `createdAt`, `triggeredBy`
3. **Viewer:** Current authenticated user
    - Fields: `username`, `permissions`

Migration Planning - Detailed Concepts

1. Why Migrate to OpenTofu?
2. Compatibility Matrix
3. Dependency Identification
4. Migration Checklist Development
5. Implementation Steps

Why Migrate to OpenTofu?

The Licensing Shift

What Happened?

In August 2023, HashiCorp changed Terraform's license from the Mozilla Public License v2.0 (MPL 2.0) to the Business Source License v1.1 (BSL 1.1). This change had significant implications for the infrastructure-as-code community.

Key License Differences:

Aspect	MPL 2.0 (Old)	BSL 1.1 (New)
Type	True open source	Source-available
Commercial Use	Unrestricted	Restricted for competitors
Derivative Works	Allowed freely	Prohibited for competing products
Community Governance	Open	Corporate controlled
Future Certainty	Predictable	Subject to change

What BSL 1.1 Means for Users:

The Business Source License prohibits using Terraform to create competing offerings. While most infrastructure teams are safe, the license creates uncertainty:

- **Direct Impact**: Companies building Terraform-as-a-Service platforms
- **Indirect Impact**: Organizations concerned about future restrictions
- **Strategic Risk**: Dependency on a single vendor's licensing decisions
- **Vendor Lock-in**: Reduced leverage in negotiations with HashiCorp

Open-Source Continuity

The OpenTofu Initiative

OpenTofu emerged as a community response to Terraform's licensing change, committed to:

1. **True Open Source**: MPL 2.0 license permanently
2. **Community Governance**: Linux Foundation management
3. **Backward Compatibility**: Drop-in replacement for Terraform
4. **Transparent Development**: Public roadmap and decision-making
5. **Vendor Neutrality**: No single company control

Who's Behind OpenTofu?

- **Linux Foundation**: Neutral governance body
- **Community Contributors**: Thousands of developers
- **Enterprise Sponsors**: Spacelift, env0, Scalr, Gruntwork, and others
- **Cloud Providers**: Support from major cloud platforms

Long-term Sustainability:

OpenTofu's governance model ensures:

- No surprise licensing changes
- Community-driven feature development
- Multiple commercial support options
- Reduced vendor dependency risk

Business Risk Mitigation

Strategic Considerations for Migration:

1. Licensing Risk Reduction

- Remove dependency on proprietary licensing terms
- Eliminate concerns about future license changes
- Maintain full commercial use rights
- Avoid potential compliance issues

2. Vendor Independence

- Multiple commercial support options available
- No single point of control
- Competitive marketplace for tools and services
- Reduced negotiation leverage concerns

3. Community Investment

- Contributions benefit entire community
- Transparent feature development
- Shared maintenance burden
- Collaborative problem-solving

4. Future-Proofing

- Guaranteed open-source access
- Community continuity regardless of corporate changes
- Multiple implementation options
- Reduced technical debt risk

5. Cost Considerations

- No licensing fees for OpenTofu
- Competitive ecosystem for commercial support
- Open marketplace for tools and integrations
- Reduced vendor lock-in costs

Feature Parity and Development

Current State (OpenTofu 1.6+):

OpenTofu maintains feature parity with Terraform v1.5.x and adds:

- All core Terraform functionality
- Improved error messages and debugging
- Enhanced state encryption options
- Community-requested features
- Performance optimizations

Future Development:

OpenTofu's roadmap includes:

- Enhanced module testing capabilities
- Improved provider development experience
- Additional state backend options
- Performance improvements
- Community-driven feature additions

Migration Timing Considerations:

Migrate Now If:

- Concerned about licensing uncertainty
- Value open-source governance
- Want to support community development
- Planning new infrastructure projects
- Updating existing tooling anyway

Wait If:

- Using Terraform Cloud features heavily (consider alternatives)
- Dependent on brand-new Terraform features (check compatibility)
- Organization has strict change control (plan carefully)
- Terraform Enterprise with specific version requirements

Compatibility Matrix

Version Compatibility

Critical Compatibility Facts:

OpenTofu 1.6.0+ is designed as a drop-in replacement for Terraform v1.5.x and earlier. Understanding the compatibility matrix is essential for migration planning.

Compatibility Table:

Component	Terraform	OpenTofu	Compatibility Status
HCL Syntax	All versions	v1.6.0+	✅ 100% compatible
State Format	v1.5.x and earlier	v1.6.0+	✅ Fully compatible
Providers	All versions	v1.6.0+	✅ Works with all providers
Modules	All versions	v1.6.0+	✅ No changes needed
Backend Configs	All versions	v1.6.0+	✅ Same syntax
CLI Commands	v1.x	v1.x	✅ Identical interface

HCL (HashiCorp Configuration Language) Compatibility

100% Backward Compatible

All valid Terraform HCL configuration files work identically with OpenTofu. No syntax changes required.

```hcl
# This configuration works identically in Terraform and OpenTofu
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = var.instance_type

  tags = {
    Name        = "web-server"
    Environment = var.environment
    ManagedBy   = "OpenTofu"  # Can update this, but not required
  }
}

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

  validation {
    condition     = contains(["t2.micro", "t2.small", "t3.micro"], var.instance_type)
    error_message = "Instance type must be t2.micro, t2.small, or t3.micro."
  }
}
```

Advanced Features Compatibility:

All Terraform HCL features work in OpenTofu:

- ✅ Dynamic blocks
- ✅ For expressions
- ✅ Conditional expressions
- ✅ Function calls
- ✅ Type constraints
- ✅ Validation rules
- ✅ Sensitive values
- ✅ Lifecycle rules
- ✅ Provisioners
- ✅ Local values

Provider Compatibility

Provider Sources Continue to Work

OpenTofu maintains compatibility with all Terraform providers, including those from the HashiCorp registry.

Provider Source Resolution:

```hcl
terraform {
  required_providers {
    # Both HashiCorp and OpenTofu registries work
    aws = {
      source  = "hashicorp/aws"  # ✅ Works perfectly
      version = "~> 5.0"
    }

    random = {
      source  = "hashicorp/random"  # ✅ Also works
      version = "~> 3.5"
    }

    # OpenTofu registry (optional, identical providers)
    custom = {
      source  = "opentofu/custom"  # ✅ OpenTofu registry
      version = "~> 1.0"
    }
  }
}
```

Provider Registry Behavior:

1. **Default Registry**: OpenTofu uses HashiCorp's provider registry by default
2. **No Changes Needed**: Existing `source` declarations work as-is
3. **Mirror Support**: Can configure provider mirrors if desired
4. **Version Locking**: `.terraform.lock.hcl` works identically

Important Notes:

- No need to change provider sources during migration
- Provider plugins are identical (same binaries work)
- Provider version constraints work the same way
- Provider authentication methods unchanged

State File Compatibility

State Format is Identical

The `.tfstate` file format is 100% compatible between Terraform and OpenTofu.

```
{
  "version": 4,
  "terraform_version": "1.5.7",  // This field may show different versions
  "serial": 1,
  "lineage": "a3c5e8f1-8c7a-4d9b-b2e6-1f5d7c9a3b4e",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "instance_type": "t2.micro"
          }
        }
      ]
    }
  ]
}
```

State Migration Process:

When you run `tofu init -migrate-state`:

1. **Reads Existing State**: OpenTofu reads the Terraform state file
2. **Validates Format**: Confirms state format compatibility
3. **Preserves Data**: Maintains all resource tracking information
4. **Updates Metadata**: Updates version field to reflect OpenTofu
5. **Maintains Lineage**: Preserves state lineage for tracking

State Backend Compatibility:

All Terraform backends work with OpenTofu:

Backend Type	Compatibility	Notes
Local	✅ Perfect	Default, no changes needed
S3	✅ Perfect	Same bucket configuration
Azure Storage	✅ Perfect	Identical settings
GCS	✅ Perfect	Same bucket syntax
Consul	✅ Perfect	No modifications required
Kubernetes	✅ Perfect	Same secret configuration
HTTP	✅ Perfect	REST API unchanged
Terraform Cloud	⚠️ Limited	Use alternative (Spacelift, env0, Scalr)

Module Compatibility

Modules Work Without Modification

All Terraform modules work identically with OpenTofu.

```
module "vpc" {
  # Local modules
  source = "./modules/vpc"                                          #   Works perfectly
}

module "security" {
  # Terraform Registry
  source  = "terraform-aws-modules/security-group/aws"              #   Works perfectly
  version = "5.1.0"
}

module "network" {
  # Git sources
  source = "git::https://github.com/org/terraform-modules.git//networking?ref=v1.2.0"          #   Works
}

module "storage" {
  # GitHub shorthand
  source = "github.com/org/modules//storage"                        #   Works
}

module "compute" {
  # S3 bucket
  source = "s3::https://s3-eu-west-1.amazonaws.com/bucket/modules/compute.zip"          #   Works
}
```

Publishing Modules:

You can publish modules to:

- **Terraform Registry**: Still works with OpenTofu consumers
- **OpenTofu Registry**: Designed for OpenTofu (same format)
- **Private Registries**: Organization-specific module hosting
- **Git Repositories**: Direct source references

CLI Command Compatibility

Identical Command-Line Interface

OpenTofu uses `tofu` instead of `terraform`, but all commands and flags work identically.

Command Mapping:

Terraform Command	OpenTofu Command	Notes
`terraform init`	`tofu init`	Identical behavior
`terraform plan`	`tofu plan`	Same options
`terraform apply`	`tofu apply`	Same workflow
`terraform destroy`	`tofu destroy`	Identical
`terraform output`	`tofu output`	Same format
`terraform state`	`tofu state`	All subcommands work
`terraform import`	`tofu import`	Identical syntax
`terraform validate`	`tofu validate`	Same validation
`terraform fmt`	`tofu fmt`	Identical formatting

Migration Simplicity Example:

bash

```
# Before migration (Terraform)
terraform init
terraform plan
terraform apply

# After migration (OpenTofu) - literally just change the binary name
tofu init
tofu plan
tofu apply
```

Dependency Identification

Identifying Module Dependencies

Why Module Identification Matters

Understanding your module dependencies is critical for:

- Ensuring continued functionality after migration
- Planning module registry updates
- Validating version constraints
- Documenting infrastructure architecture

Discovering Module Dependencies:

1. Configuration File Analysis

bash

```bash
# Find all module blocks in your configuration
grep -r "^module " . --include="*.tf"

# Find module sources
grep -r "source\s*=" . --include="*.tf" | grep "module"
```

Discovering Module Dependencies:

2. Review Each Module Declaration

hcl

```hcl
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"  # Public registry module
  version = "5.1.2"                          # Version constraint

  # Note module configuration for validation testing
  name = "my-vpc"
  cidr = "10.0.0.0/16"
}

module "custom_networking" {
  source = "./modules/networking"  # Local module (ensure code is present

  vpc_id = module.vpc.vpc_id
}

module "private_module" {
  source = "git::https://github.com/myorg/private-modules.git//networking?ref=v2.1.0"
  # Git module (ensure access credentials work)
}
```

3. Document Module Inventory

Create a module dependency matrix:

Module Name	Source Type	Source Location	Version	Used By	Critical?
vpc	Registry	terraform-aws-modules/vpc/aws	5.1.2	main.tf	Yes
networking	Local	./modules/networking	N/A	main.tf	Yes
private_net	Git	github.com/org/modules	v2.1.0	network.tf	Yes
monitoring	Registry	terraform-aws-modules/cloudwatch	4.0.1	monitoring.tf	No

Provider Dependency Mapping

Provider Version Constraints

Identify all providers and their version constraints:

1. Locate Provider Declarations

bash

```bash
# Find all required_providers blocks
grep -A 10 "required_providers" . -r --include="*.tf"
```

2. Document Provider Requirements

hcl

```hcl
terraform {
  required_version = ">= 1.5.0"  # Minimum Terraform version

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"  # Document this constraint
    }

    random = {
      source  = "hashicorp/random"
      version = ">= 3.5.0"  # And this one
    }

    tls = {
      source  = "hashicorp/tls"
      version = "~> 4.0"  # And this one
    }
  }
}
```

3. Provider Inventory Table

Provider	Source	Version Constraint	Purpose	Configuration Needs
aws	hashicorp/aws	~> 5.0	Primary cloud	Region, credentials
random	hashicorp/random	>= 3.5.0	Random values	None
tls	hashicorp/tls	~> 4.0	SSH keys, certs	None

4. Validate Provider Availability

All HashiCorp providers work with OpenTofu. Check that:

- Provider versions are available in the registry
- Version constraints are not too restrictive
- Provider configuration will work post-migration

Remote State Backend Dependencies

State Backend Configuration Review

1. Identify Current Backend

bash

```bash
# Check for backend configuration
grep -A 10 "backend \"" . -r --include="*.tf"
```

2. Document Backend Settings

hcl

```hcl
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"      # S3 bucket name
    key            = "prod/vpc/terraform.tfstate"  # State file path
    region         = "us-east-1"               # AWS region
    encrypt        = true                      # Encryption enabled
    dynamodb_table = "terraform-locks"         # Lock table

    # Note these for post-migration validation
  }
}
```

3. Backend Dependency Checklist

- [ ] Backend type identified (S3, Azure Storage, GCS, etc.)
- [ ] Access credentials documented
- [ ] Encryption settings noted
- [ ] Lock mechanism identified
- [ ] Backup procedure confirmed
- [ ] Alternative backends evaluated (if using Terraform Cloud)

4. Special Considerations for Terraform Cloud

If using Terraform Cloud backend:

```hcl
terraform {
  backend "remote" {
    organization = "my-org"

    workspaces {
      name = "production"
    }
  }
}
```

Migration Options:

- **Option 1**: Migrate to equivalent service (Spacelift, env0, Scalr)
- **Option 2**: Switch to standard remote backend (S3, GCS, Azure)
- **Option 3**: Use local backend temporarily for testing

Third-Party Tool Integrations

Identify Ecosystem Dependencies

1. CI/CD Pipeline Tools

Document tools that execute Terraform commands:

Tool	Integration Type	Changes Needed
GitHub Actions	Workflow uses terraform CLI	Update binary name to `tofu`
GitLab CI	Pipeline script calls terraform	Update commands
Jenkins	Terraform plugin or CLI	Update plugin or scripts
CircleCI	Docker image with terraform	Use OpenTofu image
Azure DevOps	Terraform task	Replace with script task

Example GitHub Actions Change:

```yaml
# Before
- name: Terraform Plan
  run: terraform plan

# After
- name: OpenTofu Plan
  run: tofu plan
```

2. Infrastructure Platforms

Platform	Integration	Migration Path
Spacelift	Native OpenTofu support	Enable OpenTofu in stack settings
env0	Native OpenTofu support	Switch tool version in environment
Atlantis	OpenTofu compatible	Update configuration
Terraform Cloud	Proprietary	Migrate to alternative

3. Development Tools

Tool	Purpose	Migration Impact
VS Code Terraform Extension	Syntax highlighting, completion	Works with OpenTofu
IntelliJ Terraform Plugin	IDE integration	Works with OpenTofu
TFLint	Linter	Works with OpenTofu
Terraform-docs	Documentation generator	Works with OpenTofu
Checkov	Security scanning	Works with OpenTofu
Infracost	Cost estimation	OpenTofu support available

4. Monitoring and Observability

Tool	Purpose	Compatibility
Datadog Terraform Integration	Monitoring	Works with OpenTofu
Terraform Compliance	Policy as code	Compatible
OPA/Sentinel	Policy enforcement	Works with OpenTofu

Migration Checklist Development

1. Environment Assessment

Infrastructure Inventory:

- [ ] List all Terraform projects/workspaces
- [ ] Identify Terraform version for each project
- [ ] Document provider versions
- [ ] Map module dependencies
- [ ] List remote state backends
- [ ] Identify shared modules

Example Inventory Spreadsheet:

Project Name	Terraform Version	Providers	Modules	Backend	Priority
prod-vpc	1.5.7	aws, random	vpc, networking	S3	Critical
dev-compute	1.5.5	aws, tls	ec2, monitoring	S3	Medium
staging-db	1.5.7	aws, postgresql	rds, backup	S3	High

2. Compatibility Verification

Version Compatibility Check:

- [ ] Confirm Terraform version is 1.5.x or earlier
- [ ] Verify provider versions are current
- [ ] Check module versions for compatibility
- [ ] Review HCL syntax for deprecated features
- [ ] Validate state file format version

Validation Commands:

```bash
# Check Terraform version
terraform version

# Verify providers
terraform providers

# Validate configuration
terraform validate

# Check state file version
grep "version" terraform.tfstate | head -1
```

3. Backup Strategy

State File Backup:

- [ ] Create backup of `.tfstate` file
- [ ] Backup `.tfstate.backup` if exists
- [ ] Document state file location (local or remote)
- [ ] Verify backup integrity
- [ ] Store backups in version control or secure storage
- [ ] Document backup restoration procedure

Backup Commands:

```
# Local state backup
cp terraform.tfstate terraform.tfstate.pre-migration-$(date +%Y%m%d)
cp terraform.tfstate.backup terraform.tfstate.backup.pre-migration-$(date +%Y%m%d)

# Remote state backup (S3 example)
aws s3 cp s3://my-bucket/terraform.tfstate ./backups/terraform.tfstate.pre-migration-$(date +%Y%m%d)
```

Backup Commands:

```
# Create backup directory with all files
tar -czf terraform-backup-$(date +%Y%m%d).tar.gz .terraform terraform.tfstate* *.tf *.tfvars
```

Configuration Backup:

- [ ] Commit all configuration files to Git
- [ ] Tag repository with pre-migration version
- [ ] Create feature branch for migration
- [ ] Document Git commit hash
- [ ] Archive configuration files separately

```
# Git workflow
git add .
git commit -m "Pre-migration snapshot"
git tag -a pre-migration-$(date +%Y%m%d) -m "Snapshot before OpenTofu migration"
git push origin --tags

# Create migration branch
git checkout -b opentofu-migration
```

4. Documentation Requirements

Document Current State:

- [ ] Current infrastructure diagram
- [ ] Resource inventory
- [ ] Dependencies map
- [ ] Provider configuration
- [ ] Backend configuration
- [ ] Workspace structure
- [ ] Environment variables

```
# Migration Plan: [Project Name]

## Current State
- Terraform Version: 1.5.7
- Infrastructure: VPC, EC2, RDS
- State Backend: S3 (bucket: my-terraform-state)
- Last Applied: 2024-01-15

## Migration Timeline
- Planning: Day 1
- Testing: Day 2-3
- Execution: Day 4
- Validation: Day 5

## Rollback Criteria
- State file corruption
- Resource drift detected
- Provider compatibility issues
- Critical errors in plan

## Success Criteria
- `tofu plan` shows no changes
- All resources in state match actual
- Outputs match expected values
- Integration tests pass
```

5. Testing Environment Setup

Create Safe Testing Environment:

- [ ] Clone production infrastructure to dev/test
- [ ] Create isolated state file
- [ ] Test migration in non-production first
- [ ] Document test results
- [ ] Validate rollback procedures

Test Environment Checklist:

- [ ] Same Terraform version as production
- [ ] Equivalent infrastructure complexity
- [ ] Separate AWS account or region
- [ ] Independent state backend
- [ ] Test credentials configured

Migration Execution Phase

1. OpenTofu Installation

Installation Steps:

- [ ] Download OpenTofu binary or use package manager
- [ ] Verify binary checksum
- [ ] Install OpenTofu (system or user level)
- [ ] Verify installation with `tofu version`
- [ ] Update PATH if necessary
- [ ] Test OpenTofu with simple configuration

```
# Verify OpenTofu installation
tofu version
# Expected output: OpenTofu v1.6.0 (or higher)

# Verify Terraform still available (for rollback)
terraform version
# Expected output: Terraform v1.5.x
```

2. State Migration

Migration Steps:

- [ ] Navigate to project directory
- [ ] Remove Terraform lock file (will be regenerated)
- [ ] Run `tofu init -migrate-state`
- [ ] Answer 'yes' to migration prompts
- [ ] Verify new lock file created
- [ ] Confirm state migration in output

```
# Remove old lock file (will be regenerated)
rm .terraform.lock.hcl

# Initialize with OpenTofu and migrate state
tofu init -migrate-state
# Expected prompt: "Do you want to migrate state from Terraform to OpenTofu?"
# Answer: yes

# Verify lock file regenerated
ls -la .terraform.lock.hcl
```

3. Plan Validation

Validation Steps:

- [ ] Run `tofu plan` to generate new plan
- [ ] Review plan output thoroughly
- [ ] Verify plan shows zero changes
- [ ] Check for any warnings or notices
- [ ] Save plan output for comparison
- [ ] Compare with pre-migration plan

```
# Generate plan with OpenTofu
tofu plan -out=post-migration.plan

# Expected output should show:
# "No changes. Your infrastructure matches the configuration."

# If changes are shown, investigate before proceeding:
# - Resource replacements indicate issues
# - Attribute changes may indicate provider differences
# - Output changes are generally harmless
```

4. Resource Verification

Verification Checklist:

- [ ] All resources present in state
- [ ] Resource counts match expectations
- [ ] No unexpected additions or removals
- [ ] Dependencies correctly mapped
- [ ] Outputs generate correctly

```
# List all resources
tofu state list

# Show specific resource details
tofu state show aws_vpc.main

# Verify outputs
tofu output

# Check resource count
tofu state list | wc -l
```

Post-Migration Phase

1. State Consistency Validation

Validation Steps:

- [ ] Run `tofu plan` again
- [ ] Confirm zero changes shown
- [ ] Validate remote state if applicable
- [ ] Check state lock functionality
- [ ] Verify state backup created
- [ ] Test state refresh

```
# Refresh state from actual infrastructure
tofu refresh

# Plan should still show no changes
tofu plan

# For remote state, verify backend access
tofu state pull > current-state.json
cat current-state.json | jq '.serial'  # Should match expected version
```

2. Module Registry Validation

Registry Check:

- [ ] Verify all modules still accessible
- [ ] Check module versions match constraints
- [ ] Test module initialization
- [ ] Validate private registry access if applicable
- [ ] Confirm module outputs still work

Module Validation:

```
# Re-initialize to verify module access
rm -rf .terraform/modules
tofu init

# Should download all modules successfully
# Check module versions
cat .terraform/modules/modules.json | jq '.Modules[] | {key: .Key, source: .Source, version: .Version}'
```

3. Provider Functionality Testing

Provider Tests:

- [ ] Verify provider authentication works
- [ ] Test resource creation (in test environment)
- [ ] Test resource updates
- [ ] Test resource deletion
- [ ] Validate provider-specific features
- [ ] Check provider version constraints

```
# In test environment, create a simple resource
cat > test.tf << 'EOF'
resource "random_pet" "test" {
  length = 2
}

output "pet_name" {
  value = random_pet.test.id
}
EOF

tofu init
tofu apply -auto-approve
tofu destroy -auto-approve
```

4. Reference and Documentation Updates

Update Documentation:

- [ ] Update README files (terraform → tofu)
- [ ] Modify runbooks and procedures
- [ ] Update CI/CD documentation
- [ ] Change team wiki/knowledge base
- [ ] Update onboarding materials
- [ ] Revise troubleshooting guides

```
# Infrastructure Management

<!-- Before -->
## Prerequisites
- Terraform v1.5+
- AWS CLI configured

## Usage
terraform init
terraform plan
terraform apply

<!-- After -->
## Prerequisites
- OpenTofu v1.6+
- AWS CLI configured

## Usage
tofu init
tofu plan
tofu apply
```

Update CI/CD References:

- [ ] GitHub Actions workflows
- [ ] GitLab CI pipelines
- [ ] Jenkins jobs
- [ ] CircleCI configurations
- [ ] Azure DevOps pipelines
- [ ] Custom scripts and tools

```yaml
# .github/workflows/infrastructure.yml

# Before
- name: Setup Terraform
  uses: hashicorp/setup-terraform@v2
  with:
    terraform_version: 1.5.7

- name: Terraform Init
  run: terraform init

# After
- name: Setup OpenTofu
  uses: opentofu/setup-opentofu@v1
  with:
    tofu_version: 1.6.0

- name: OpenTofu Init
  run: tofu init
```

5. Team Communication

Communication Checklist:

- [ ] Notify team of migration completion
- [ ] Share migration results and findings
- [ ] Document lessons learned
- [ ] Update team procedures
- [ ] Schedule knowledge sharing session
- [ ] Provide migration support resources

```
# OpenTofu Migration Completion Report

## Project: [Project Name]
## Date: [Date]
## Performed By: [Name]

### Summary
Successfully migrated [Project Name] from Terraform v1.5.7 to OpenTofu v1.6.0.

### Results
- ✅ State migration completed without errors
- ✅ Zero infrastructure changes detected
- ✅ All modules functioning correctly
- ✅ CI/CD pipelines updated
- ✅ Documentation revised

### Validation
- Plan output: No changes
- Resource count: X resources unchanged
- State serial: Incremented correctly
- Outputs: All verified

### Issues Encountered
- [None | List any issues and resolutions]

### Next Steps
- Team training scheduled for [date]
- Monitoring for one week
- Additional projects to migrate: [list]

### Rollback Plan
- State backup location: [path/URL]
- Rollback tested: [Yes/No]
- Rollback time estimate: [X minutes]
```

Rollback Procedures

When to Rollback:

- State file corruption detected
- Significant infrastructure drift
- Provider compatibility issues
- Critical functionality broken
- Team decision to postpone

Rollback Checklist:

- [ ] Stop any active operations
- [ ] Restore state file from backup
- [ ] Restore configuration from Git tag
- [ ] Reinitialize with Terraform
- [ ] Validate restoration with `terraform plan`
- [ ] Document rollback reason
- [ ] Plan remediation

Rollback Commands:

bash

```
# Stop current operations
# Press Ctrl+C if anything running

# Restore state from backup
cp terraform.tfstate.pre-migration terraform.tfstate
cp terraform.tfstate.backup.pre-migration terraform.tfstate.backup

# Restore configuration from Git
git checkout pre-migration-$(date +%Y%m%d)

# Remove OpenTofu artifacts
rm -rf .terraform .terraform.lock.hcl

# Reinitialize with Terraform
terraform init

# Verify restoration
terraform plan
# Should show same state as before migration
```

Summary

This migration planning tutorial covers the essential knowledge needed to successfully migrate from Terraform to OpenTofu:

1. **Understanding Why**: Licensing changes, open-source continuity, and business risk mitigation
2. **Compatibility Assurance**: HCL, providers, state files, and modules all work identically
3. **Dependency Mapping**: Identifying all components that need validation during migration
4. **Systematic Approach**: Comprehensive checklists for every phase of migration

Hands-on Migration Lab - VPC + EC2 Migration

Lab Overview

Objective: Migrate a production-like AWS infrastructure project from Terraform to OpenTofu, validating state consistency and resource integrity throughout the process.

Infrastructure Components:
Multi-AZ VPC with public and private subnets
Internet Gateway and NAT Gateway
Route tables and associations
EC2 instance with SSH access
Security groups with minimal access
Elastic IP for NAT Gateway
SSH key pair for instance access

Learning Outcomes:

Experience a complete migration workflow
Validate state consistency before and after migration
Troubleshoot common migration issues
Document migration results
Understand rollback procedures