**🔹 count in Terraform**

* count is used when you want to create **multiple identical resources**.
* Works well with **lists** (not maps).
* Resource instances are **indexed numerically** (0, 1, 2…).

**✅ Example: Create Multiple AWS EC2 Instances with count**

variable "instance\_count" {

default = 3

}

resource "aws\_instance" "ec2\_example" {

count = var.instance\_count

ami = "ami-0abcdef1234567890"

instance\_type = "t2.micro"

tags = {

Name = "web-server-${count.index + 1}"

}

}

👉 This will create **3 EC2 instances** named web-server-1, web-server-2, and web-server-3.  
🔹 Use case: Scaling multiple servers with the same configuration.

**🔹 for\_each in Terraform**

* for\_each is used when resources need **unique configurations**.
* Works well with **maps or sets**.
* Resource instances are identified by **map keys** instead of index numbers.

**✅ Example: Create EC2 Instances with Different Names and Types**

variable "instances" {

default = {

"app1" = "t2.micro"

"app2" = "t2.small"

"app3" = "t3.micro"

}

}

resource "aws\_instance" "ec2\_example" {

for\_each = var.instances

ami = "ami-0abcdef1234567890"

instance\_type = each.value

tags = {

Name = each.key

}

}

👉 This creates **3 EC2s**:

* app1 → t2.micro
* app2 → t2.small
* app3 → t3.micro

🔹 Use case: When every resource needs **different parameters**.

**🔹 Real-Time AWS Use Cases**

**1. count Use Cases**

* Multiple **subnets** in the same VPC:

resource "aws\_subnet" "public\_subnet" {

count = 3

vpc\_id = aws\_vpc.main.id

cidr\_block = cidrsubnet(aws\_vpc.main.cidr\_block, 8, count.index)

availability\_zone = element(data.aws\_availability\_zones.available.names, count.index)

tags = {

Name = "public-subnet-${count.index + 1}"

}

}

👉 Creates 3 subnets across AZs.

* Multiple **identical IAM users** (lab environment).

**2. for\_each Use Cases**

* Creating **S3 Buckets** with different names and tags:

variable "buckets" {

default = {

"logs" = "Environment=dev"

"backup" = "Environment=prod"

}

}

resource "aws\_s3\_bucket" "buckets" {

for\_each = var.buckets

bucket = "${each.key}-bucket-demo"

tags = {

Name = each.key

Environment = each.value

}

}

👉 Creates two buckets:

* logs-bucket-demo (dev)
* backup-bucket-demo (prod)
* Multiple **IAM roles/policies** with different permissions.
* **Security groups** with different inbound rules per application.

**🔹 When to Use count vs for\_each**

| **Feature** | **count** | **for\_each** |
| --- | --- | --- |
| Best for | Identical resources | Resources with unique configs |
| Indexing | Numeric (count.index) | Key-based (each.key, each.value) |
| Works with | List/number | Map/set |
| Example | 3 EC2s with same type | 3 EC2s with different names/types |

👉 **Rule of Thumb**:

* Use **count** when the resources are **identical** (just want N copies).
* Use **for\_each** when the resources need **unique configurations** (map of values).

Do you want me to also prepare a **side-by-side AWS EC2 example with both count and for\_each** so you can directly compare the Terraform plans?

Let’s do a **side-by-side AWS EC2 example** using both **count** and **for\_each** so you can clearly see the difference in resource creation and Terraform plan output.

**🔹 Example 1: Using count (Identical Instances)**

variable "instance\_count" {

default = 3

}

resource "aws\_instance" "ec2\_count\_example" {

count = var.instance\_count

ami = "ami-0abcdef1234567890"

instance\_type = "t2.micro"

tags = {

Name = "web-server-${count.index + 1}"

}

}

**✅ What Happens:**

* Terraform will create **3 EC2 instances**, all with:
  + AMI: ami-0abcdef1234567890
  + Instance type: t2.micro
* Instance names:
  + web-server-1
  + web-server-2
  + web-server-3

👉 Use case: You need **identical EC2 servers** (like scaling a web tier).

**🔹 Example 2: Using for\_each (Unique Configs per Instance)**

variable "instances" {

default = {

"app1" = "t2.micro"

"app2" = "t2.small"

"app3" = "t3.micro"

}

}

resource "aws\_instance" "ec2\_foreach\_example" {

for\_each = var.instances

ami = "ami-0abcdef1234567890"

instance\_type = each.value

tags = {

Name = each.key

}

}

**✅ What Happens:**

* Terraform will create **3 EC2 instances**, but with **different configurations**:
  + app1 → t2.micro
  + app2 → t2.small
  + app3 → t3.micro

👉 Use case: You need **different instance types/names** for each application.

**🔹 Key Differences in Plan Output**

**count**

Terraform plan will show resources as:

aws\_instance.ec2\_count\_example[0]

aws\_instance.ec2\_count\_example[1]

aws\_instance.ec2\_count\_example[2]

**for\_each**

Terraform plan will show resources as:

aws\_instance.ec2\_foreach\_example["app1"]

aws\_instance.ec2\_foreach\_example["app2"]

aws\_instance.ec2\_foreach\_example["app3"]

**✅ Quick Decision Table**

| **Situation** | **Use** |
| --- | --- |
| I need N identical resources | **Count** |
| I need uniquely configured resources | **for\_each** |
| Want to scale with a number | **Count** |
| Want to reference by name (not index) | **for\_each** |

⚡ So in real AWS projects:

* **Web server ASG / Subnets → count**
* **S3 buckets / IAM policies / Security groups with different rules → for\_each**

Let’s build a **Terraform module for EC2** that supports both **count** and **for\_each**, so you can switch based on your use case without rewriting everything.

**📦 Terraform Module: ec2-instance**

**modules/ec2-instance/main.tf**

variable "use\_for\_each" {

type = bool

default = false

}

variable "instances" {

description = "Map of instance\_name => instance\_type (used when for\_each = true)"

type = map(string)

default = {}

}

variable "instance\_count" {

description = "Number of instances (used when for\_each = false)"

type = number

default = 1

}

variable "ami" {

type = string

default = "ami-0abcdef1234567890"

}

# --- EC2 with for\_each ---

resource "aws\_instance" "ec2\_foreach" {

for\_each = var.use\_for\_each ? var.instances : {}

ami = var.ami

instance\_type = each.value

tags = {

Name = each.key

}

}

# --- EC2 with count ---

resource "aws\_instance" "ec2\_count" {

count = var.use\_for\_each ? 0 : var.instance\_count

ami = var.ami

instance\_type = "t2.micro"

tags = {

Name = "web-server-${count.index + 1}"

}

}

**🔹 How to Use the Module**

**Option 1: Using count**

module "ec2\_count\_example" {

source = "./modules/ec2-instance"

use\_for\_each = false

instance\_count = 3

ami = "ami-0abcdef1234567890"

}

👉 Creates **3 identical EC2s**:

* web-server-1
* web-server-2
* web-server-3

**Option 2: Using for\_each**

module "ec2\_foreach\_example" {

source = "./modules/ec2-instance"

use\_for\_each = true

ami = "ami-0abcdef1234567890"

instances = {

"app1" = "t2.micro"

"app2" = "t2.small"

"app3" = "t3.micro"

}

}

👉 Creates **3 EC2s with unique configs**:

* app1 → t2.micro
* app2 → t2.small
* app3 → t3.micro

**✅ Why This Helps**

* You don’t have to maintain two separate Terraform configs.
* Just flip the use\_for\_each flag depending on the project.
* Works great in **multi-environment AWS setups** (Dev = count, Prod = for\_each).

⚡ Next Step: I can also add **outputs** to return instance IDs, public IPs, and tags for both modes — so the module is fully reusable.

**Terraform Meta-Arguments**

Meta-arguments are **special arguments** that can be used with **any resource/module** in Terraform to control **how resources are created, destroyed, or managed**.

The most important ones are:

* count
* for\_each
* depends\_on
* provider
* lifecycle

**1. count**

👉 Creates **multiple instances** of a resource.

**Example:**

Suppose you want **3 EC2 instances** in AWS.

resource "aws\_instance" "web" {

count = 3

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

tags = {

Name = "web-${count.index}"

}

}

✅ **Real-time use case**:

* Creating N number of EC2 instances for a web server pool.
* Scaling test environments by number of instances.

**2. for\_each**

👉 Creates **resources for each key/value** in a map or set.  
More flexible than count (works well with named resources).

**Example:**

Creating EC2 instances for **different environments**.

resource "aws\_instance" "env" {

for\_each = {

dev = "t2.micro"

test = "t2.small"

prod = "t3.medium"

}

ami = "ami-0c55b159cbfafe1f0"

instance\_type = each.value

tags = {

Name = "${each.key}-server"

}

}

✅ **Real-time use case**:

* Creating EC2s with **different sizes per environment**.
* Creating S3 buckets per department (HR, Finance, Dev).

**3. depends\_on**

👉 Explicitly tells Terraform about **dependencies** (when not obvious).

**Example:**

You want to create an EC2 **only after** an S3 bucket is created.

resource "aws\_s3\_bucket" "app\_bucket" {

bucket = "my-app-bucket"

}

resource "aws\_instance" "app" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

depends\_on = [aws\_s3\_bucket.app\_bucket]

}

✅ **Real-time use case**:

* Ensuring DB subnet group is created before RDS instance.
* Making sure IAM role is created before EC2 instance attaches it.

**4. provider**

👉 Lets you **use multiple providers** (regions/accounts).

**Example:**

Deploy EC2 in **two different AWS regions**.

provider "aws" {

alias = "us\_east"

region = "us-east-1"

}

provider "aws" {

alias = "us\_west"

region = "us-west-2"

}

resource "aws\_instance" "east" {

provider = aws.us\_east

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

resource "aws\_instance" "west" {

provider = aws.us\_west

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

✅ **Real-time use case**:

* Multi-region deployments (DR setup).
* Deploying infra in multiple AWS accounts.

**5. lifecycle**

👉 Controls **create/destroy behavior** of resources.

**Example 1: Prevent accidental deletion**

resource "aws\_s3\_bucket" "logs" {

bucket = "critical-logs-bucket"

lifecycle {

prevent\_destroy = true

}

}

✅ Use case: Protecting **production S3 buckets or RDS databases**.

**Example 2: Replace before destroy**

resource "aws\_instance" "db" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.medium"

lifecycle {

create\_before\_destroy = true

}

}

✅ Use case: Zero-downtime upgrades of EC2/RDS.

**Example 3: Ignore changes in specific attributes**

resource "aws\_instance" "web" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

lifecycle {

ignore\_changes = [tags]

}

}

✅ Use case: If DevOps team manages infra, but developers keep updating **tags** manually — Terraform won’t revert them.

**🎯 Summary of Use Cases**

* **count** → Create N identical resources (e.g., 5 EC2s).
* **for\_each** → Create resources based on map/set (e.g., EC2 per environment).
* **depends\_on** → Control resource creation order (e.g., DB before app server).
* **provider** → Multi-region/multi-account deployments.
* **lifecycle** → Control resource replacement, deletion, or ignored changes.

Terraform Lifecycle Arguments

1. create\_before\_destroy

Ensures a new resource is created before destroying the old one.

Useful for avoiding downtime.

✅ Example: EC2 Instance Replacement

resource "aws\_instance" "web" {

ami = "ami-0abcd1234efgh5678"

instance\_type = "t2.micro"

lifecycle {

create\_before\_destroy = true

}

}

👉 When you change the AMI or instance type, Terraform first creates the new EC2 instance and then destroys the old one (avoiding downtime).

2. prevent\_destroy

Protects a resource from accidental deletion.

Useful for critical resources like production databases, S3 buckets, or VPCs.

✅ Example: RDS Instance

resource "aws\_db\_instance" "prod\_db" {

allocated\_storage = 20

engine = "mysql"

instance\_class = "db.t3.micro"

name = "prod"

username = "admin"

password = "password"

skip\_final\_snapshot = true

lifecycle {

prevent\_destroy = true

}

}

👉 Even if you run terraform destroy, Terraform will refuse to delete this DB unless you remove the lifecycle rule.

3. ignore\_changes

Ignores changes to specific attributes.

Useful when external processes (manual changes, automation tools) modify a resource property, but you don’t want Terraform to overwrite it.

✅ Example: Auto Scaling Desired Capacity

resource "aws\_autoscaling\_group" "example" {

name = "example-asg"

min\_size = 1

max\_size = 5

desired\_capacity = 2

launch\_configuration = aws\_launch\_configuration.example.id

vpc\_zone\_identifier = ["subnet-123456"]

lifecycle {

ignore\_changes = [desired\_capacity]

}

}

👉 Terraform won’t reset the desired capacity if it changes manually or due to scaling policies.

4. replace\_triggered\_by (Terraform 0.13+)

Forces a resource replacement if another resource changes.

Useful for dependent resources that need recreation on parent changes.

✅ Example: Replace EC2 when Security Group Changes

resource "aws\_security\_group" "web\_sg" {

name = "web-sg"

}

resource "aws\_instance" "web" {

ami = "ami-0abcd1234efgh5678"

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.web\_sg.id]

lifecycle {

replace\_triggered\_by = [aws\_security\_group.web\_sg]

}

}

👉 If the SG is replaced, the EC2 instance is also replaced.

📌 Summary of Lifecycle Arguments

Argument Purpose Example Use Case

create\_before\_destroy Avoid downtime by creating new before destroying old EC2, Load Balancer

prevent\_destroy Prevent accidental deletion RDS, S3 bucket, VPC

ignore\_changes Ignore changes to attributes Autoscaling desired\_capacity

replace\_triggered\_by Replace when another resource changes EC2 replaced when SG changes

⚡ Real-world DevOps Use Cases:

Production RDS → use prevent\_destroy

AutoScaling Groups → use ignore\_changes for desired capacity

Blue/Green Deployments → use create\_before\_destroy

Dependencies (SG, IAM) → use replace\_triggered\_by

**Terraform Modules — Use Cases**

**1. Reusability Across Environments**

Instead of writing the same VPC, EC2, or RDS code for **dev, test, and prod**, create a **VPC module** and reuse it.

✅ **Example: VPC Module**

# main.tf

module "vpc" {

source = "./modules/vpc"

cidr = "10.0.0.0/16"

subnets = ["10.0.1.0/24", "10.0.2.0/24"]

region = "us-east-1"

}

👉 You can reuse the same vpc module for multiple environments by passing different inputs.

**2. Standardization of Infrastructure**

Organizations use modules to enforce **best practices** and **security standards**.

* Example: Every EC2 instance must have a specific IAM role, tags, and monitoring.
* Instead of relying on engineers to remember, package these rules into a module.

✅ **Example: EC2 Standard Module**

module "ec2\_instance" {

source = "git::https://github.com/org/modules//ec2"

instance\_type = "t3.micro"

ami = "ami-123456"

monitoring = true

tags = {

Environment = "prod"

Owner = "DevOps"

}

}

**3. Multi-Cloud Deployments**

When managing resources across **AWS, Azure, GCP**, you can create cloud-specific modules.

* Example: A storage module that provisions S3 on AWS, Blob on Azure, and GCS on GCP depending on provider input.

👉 Helps companies running **hybrid or multi-cloud strategies**.

**4. Team Collaboration**

Large teams create **shared Terraform registries** with common modules (network, compute, security).

* Dev teams just call the modules instead of writing everything from scratch.
* Ensures **consistency** across all projects.

**5. Disaster Recovery & DR Environments**

Modules make it easy to replicate entire infrastructure in another region for **DR setup**.

* Example: Use the same VPC, RDS, and EC2 modules in us-east-1 and us-west-2.

**6. CI/CD Pipelines with Terraform**

In DevOps pipelines, modules simplify infrastructure deployment.

* Example: Jenkins/GitHub Actions pipeline uses the same module for provisioning infrastructure for PR environments (short-lived environments).

**7. Third-Party and Open Source Modules**

Terraform **Registry** (registry.terraform.io) has official & community modules.

* Example: terraform-aws-modules/vpc/aws for VPCs.
* Saves time and ensures you follow AWS best practices.

✅ **Example: Using AWS VPC Public Module**

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "5.0.0"

name = "prod-vpc"

cidr = "10.0.0.0/16"

azs = ["us-east-1a", "us-east-1b"]

public\_subnets = ["10.0.1.0/24", "10.0.2.0/24"]

private\_subnets = ["10.0.3.0/24", "10.0.4.0/24"]

}

👉 With just a few lines, you get a **production-grade VPC** with routing, NAT, and subnets.

**📌 Summary of Terraform Module Use Cases**

| **Use Case** | **Benefit** |
| --- | --- |
| **Reusable environments** | Use same code for dev, test, prod |
| **Standardization** | Enforce org-wide security & tagging policies |
| **Multi-cloud infra** | Deploy similar resources across AWS, Azure, GCP |
| **Team collaboration** | Share modules across teams/projects |
| **Disaster recovery** | Replicate infra in another region easily |
| **CI/CD integration** | Spin up infra on demand in pipelines |
| **Use open-source modules** | Save time & follow best practices |

👉 In real-world companies, Terraform codebases are usually **80% modules, 20% custom resources**.  
This keeps things **clean, reusable, and scalable**.

Here’s a **complete guide to Terraform Modules Best Practices** with examples 👇

**🔹 Terraform Modules Best Practices**

**1. Keep Modules Small and Focused**

* A module should do **one job well** (e.g., VPC, EC2, RDS, S3).
* Don’t create “mega-modules” that handle everything at once.

✅ **Example:**

* modules/vpc → handles only VPC, subnets, routes.
* modules/ec2 → handles EC2 instance setup.

**2. Use Input Variables with Defaults**

* Expose configuration via **variables** instead of hardcoding.
* Provide sensible defaults.

✅ **Example:** variables.tf

variable "instance\_type" {

description = "EC2 instance type"

type = string

default = "t3.micro"

}

variable "tags" {

description = "Resource tags"

type = map(string)

default = {}

}

**3. Always Output Useful Information**

* Outputs make modules easier to consume in other modules/environments.

✅ **Example:** outputs.tf

output "vpc\_id" {

description = "The ID of the VPC"

value = aws\_vpc.main.id

}

output "public\_subnet\_ids" {

description = "IDs of the public subnets"

value = aws\_subnet.public[\*].id

}

**4. Version Control Modules**

* Pin module versions (version = "x.y.z") when using **Terraform Registry** or Git.
* Prevents breaking changes when upstream module updates.

✅ Example:

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "5.0.0"

}

**5. Use terraform fmt, validate, and tflint**

* Enforce formatting (terraform fmt -check).
* Validate syntax before committing.
* Use tflint to catch issues early.

**6. Document Each Module**

* Add a README.md with usage examples, inputs, and outputs.
* Helps new engineers understand how to use it.

✅ Example README snippet:

# EC2 Module

This module provisions an EC2 instance.

## Inputs

- `instance\_type` (string): EC2 type (default: `t3.micro`)

- `ami` (string): AMI ID

## Outputs

- `instance\_id`: The ID of the EC2 instance

**7. Avoid Hardcoding Provider Configurations in Modules**

* Don’t put provider "aws" inside modules.
* Keep provider settings at the **root level**, so modules remain portable.

**8. Follow a Clear Folder Structure**

✅ **Recommended layout:**

terraform-project/

├── modules/

│ ├── vpc/

│ ├── ec2/

│ └── rds/

├── environments/

│ ├── dev/

│ ├── stage/

│ └── prod/

└── main.tf

👉 Each environment (dev, stage, prod) reuses the same modules with different variables.

**9. Make Modules Idempotent**

* Modules should produce the same result **every run** if no inputs change.
* Avoid using random\_id without state handling.

**10. Don’t Overuse count or for\_each Inside Modules**

* Keep flexibility, but don’t make modules overly complex with too many loops.
* Instead, let the **root module** control how many instances are created.

**11. Tag Everything**

* Always enforce tagging in modules.
* Helps with cost tracking, auditing, and compliance.

✅ Example:

resource "aws\_instance" "web" {

ami = var.ami

instance\_type = var.instance\_type

tags = merge(

var.tags,

{ "Name" = "web-server" }

)

}

**12. Testing Modules**

* Use **Terratest (Go)** or **kitchen-terraform** to test modules.
* At minimum, run terraform plan in CI/CD pipelines for validation.

**📌 Quick Recap of Best Practices**

✅ Small, focused modules  
✅ Use input variables & outputs  
✅ Pin versions for stability  
✅ Format + lint + validate  
✅ Document with README  
✅ Keep providers outside modules  
✅ Organize with clear folder structure  
✅ Ensure idempotency  
✅ Enforce tagging & security policies  
✅ Test modules in CI/CD

**Terraform Workspaces**

A **workspace** in Terraform is an isolated environment that allows you to use the **same Terraform configuration** to manage **different environments** (like dev, stage, prod) **without duplicating code**.

By default, Terraform starts with a workspace called default.

**🔑 Key Terraform Workspace Commands**

| **Command** | **Description** |
| --- | --- |
| terraform workspace list | List all available workspaces |
| terraform workspace show | Show the current active workspace |
| terraform workspace new <name> | Create a new workspace |
| terraform workspace select <name> | Switch to an existing workspace |
| terraform workspace delete <name> | Delete a workspace |
| terraform workspace select default | Switch back to the default workspace |

**⚙️ Workspace Example: AWS Infrastructure**

Suppose you want to create **separate S3 buckets** for dev, stage, and prod using the same Terraform configuration.

**📂 Directory structure:**

main.tf

variables.tf

**main.tf**

provider "aws" {

region = "us-east-1"

}

resource "aws\_s3\_bucket" "example" {

bucket = "myapp-${terraform.workspace}-bucket"

acl = "private"

}

**variables.tf**

variable "project" {

default = "myapp"

}

**🏃 Running Terraform with Workspaces**

**1️⃣ Initialize Terraform**

terraform init

**2️⃣ Create Workspaces**

terraform workspace new dev

terraform workspace new stage

terraform workspace new prod

**3️⃣ Switch Between Workspaces**

terraform workspace select dev

terraform apply -auto-approve

👉 Creates myapp-dev-bucket

terraform workspace select stage

terraform apply -auto-approve

👉 Creates myapp-stage-bucket

terraform workspace select prod

terraform apply -auto-approve

👉 Creates myapp-prod-bucket

**📌 Use Cases of Terraform Workspaces**

1. **Multi-Environment Setup (Dev/Stage/Prod)**
   * Use the same code base but deploy resources into separate environments.
   * Example: dev, stage, prod VPCs, databases, or clusters.
2. **Isolated Testing Environments**
   * Test changes in a temporary workspace before merging into production.
   * Example: Run terraform workspace new feature-x for testing.
3. **Per-Customer Deployments in SaaS**
   * If building SaaS where each customer has isolated infra.
   * Example: terraform workspace new customer1, terraform workspace new customer2.
4. **Avoiding Multiple Codebases**
   * Instead of copying code into multiple folders for each environment, use one codebase with workspaces.

**⚠️ Workspace Limitations**

* Workspaces **do not isolate state backends** completely in remote storage.
* They are **not suitable for very large scale environment separation** (like completely different accounts).
* For enterprise setups, use **Terraform modules + separate state files per environment** instead of only workspaces.

✅ **Best Practice**:

* Use **Terraform Workspaces** for small-scale environment separation.
* For enterprise, combine **Workspaces + Remote State (S3/DynamoDB, Terraform Cloud, or Azure Storage)** + **Modules** for better isolation.

**Terraform Modules with Workspaces**

A **Terraform Module** is a reusable block of Terraform code (like a function).  
When combined with **Workspaces**, you can **reuse the same module** to deploy infra into multiple environments without duplicating files.

**📂 Example: Multi-Environment VPC using Module + Workspaces**

**1️⃣ Directory Structure**

terraform-project/

│── main.tf

│── variables.tf

│── outputs.tf

│── modules/

│ └── vpc/

│ ├── main.tf

│ ├── variables.tf

│ └── outputs.tf

**2️⃣ Module Definition (modules/vpc/main.tf)**

resource "aws\_vpc" "this" {

cidr\_block = var.vpc\_cidr

enable\_dns\_hostnames = true

tags = {

Name = "${var.project}-${terraform.workspace}-vpc"

}

}

output "vpc\_id" {

value = aws\_vpc.this.id

}

**modules/vpc/variables.tf**

variable "vpc\_cidr" {

type = string

}

variable "project" {

type = string

default = "myapp"

}

**3️⃣ Root Module (main.tf)**

provider "aws" {

region = "us-east-1"

}

# Define per-workspace variables using maps

locals {

vpc\_cidrs = {

dev = "10.0.0.0/16"

stage = "10.1.0.0/16"

prod = "10.2.0.0/16"

}

}

module "vpc" {

source = "./modules/vpc"

vpc\_cidr = local.vpc\_cidrs[terraform.workspace]

project = "myapp"

}

**4️⃣ Run with Workspaces**

# Initialize

terraform init

# Create environments

terraform workspace new dev

terraform workspace new stage

terraform workspace new prod

# Deploy Dev

terraform workspace select dev

terraform apply -auto-approve

# Creates VPC: myapp-dev-vpc with CIDR 10.0.0.0/16

# Deploy Stage

terraform workspace select stage

terraform apply -auto-approve

# Creates VPC: myapp-stage-vpc with CIDR 10.1.0.0/16

# Deploy Prod

terraform workspace select prod

terraform apply -auto-approve

# Creates VPC: myapp-prod-vpc with CIDR 10.2.0.0/16

**📌 Real-World Use Cases**

1. **Multi-Environment Deployment**
   * Use the same **VPC module** to deploy dev, stage, and prod VPCs with different CIDRs.
2. **Reusing Modules Across Teams**
   * Example: A **database module** reused for multiple environments with different sizes (db.t3.micro in dev, db.m5.large in prod).
3. **SaaS Multi-Tenant Deployments**
   * Create a workspace per customer (workspace: customer1, workspace: customer2) using the same module.

**✅ Best Practices**

* Use **modules** for reusability, **workspaces** for environment isolation.
* Store **remote state** (S3 + DynamoDB, Terraform Cloud, or Azure Storage) so each workspace has separate state.
* Avoid putting **sensitive data** inside workspace-dependent maps; instead use **TFVARS per workspace** + terraform.workspace.

**Use Conditional Logic Inside the Module**

Inside the module, wrap resources with count or for\_each based on a variable.

✅ **Example: In your modules/vpc/subnets.tf**

variable "create\_subnets" {

description = "Whether to create subnets"

type = bool

default = true

}

resource "aws\_subnet" "this" {

count = var.create\_subnets ? length(var.subnet\_cidrs) : 0

vpc\_id = aws\_vpc.this.id

cidr\_block = var.subnet\_cidrs[count.index]

availability\_zone = element(var.azs, count.index)

}

👉 Then, when calling the module:

module "vpc" {

source = "./modules/vpc"

cidr = "10.0.0.0/16"

create\_subnets = false # 🚫 No subnets created

}

**🔹 2. Use Null Values with for\_each or count**

If you pass empty lists/maps, the resources won’t be created.

✅ Example:

variable "subnet\_cidrs" {

type = list(string)

default = []

}

resource "aws\_subnet" "this" {

for\_each = { for idx, cidr in var.subnet\_cidrs : idx => cidr }

vpc\_id = aws\_vpc.this.id

cidr\_block = each.value

}

👉 If you don’t want subnets:

module "vpc" {

source = "./modules/vpc"

cidr = "10.0.0.0/16"

subnet\_cidrs = [] # 🚫 Empty list = No subnets created

}

**🔹 3. Use Module Composition Instead of a Big Module**

Instead of putting **VPC + Subnets + NAT** in one big module, split them into **separate modules** and compose them in the root.

✅ Folder structure:

modules/

vpc/

subnets/

nat/

👉 Usage:

module "vpc" {

source = "./modules/vpc"

cidr = "10.0.0.0/16"

}

# Optional — only call this if you want subnets

# module "subnets" {

# source = "./modules/subnets"

# vpc\_id = module.vpc.vpc\_id

# subnets = ["10.0.1.0/24", "10.0.2.0/24"]

# }

👉 This way, if you don’t want subnets, you just **don’t call the subnets module** at all.

**📌 Best Practice Recommendation**

* If it’s your **own custom module** → go with **Option 1 (conditional variable)**, it’s flexible.
* If you’re using a **public module (like terraform-aws-modules/vpc/aws)** → use **Option 2 (empty list/map)** since many official modules already support this pattern.
* If you’re building infra at scale → prefer **Option 3 (modular design)**, because it gives better long-term control.

Perfect ✅ let’s take the **official AWS VPC module** (terraform-aws-modules/vpc/aws) as the real-world example.

This module is super flexible — it already supports disabling subnets by passing **empty lists**.

**🔹 Example: Create Only a VPC (No Subnets)**

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "5.0.0"

name = "my-vpc"

cidr = "10.0.0.0/16"

# 🚫 Disable all subnets by passing empty lists

azs = []

private\_subnets = []

public\_subnets = []

database\_subnets = []

enable\_dns\_hostnames = true

enable\_dns\_support = true

}

👉 Result:

* VPC is created (10.0.0.0/16)
* **No subnets, no NAT, no IGW**

**🔹 Example: VPC with Public Subnets Only**

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "5.0.0"

name = "public-vpc"

cidr = "10.0.0.0/16"

azs = ["us-east-1a", "us-east-1b"]

public\_subnets = ["10.0.1.0/24", "10.0.2.0/24"]

private\_subnets = [] # 🚫 Disable private

database\_subnets = [] # 🚫 Disable DB subnets

}

👉 Result:

* VPC with only **2 public subnets**.
* No private subnets, no database subnets.

**🔹 Example: Full VPC but Disable NAT Gateway**

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "5.0.0"

name = "custom-vpc"

cidr = "10.0.0.0/16"

azs = ["us-east-1a", "us-east-1b"]

private\_subnets = ["10.0.1.0/24", "10.0.2.0/24"]

public\_subnets = ["10.0.3.0/24", "10.0.4.0/24"]

enable\_nat\_gateway = false # 🚫 Disable NAT

}

👉 Result:

* VPC with public + private subnets.
* No NAT gateway is created.

**📌 Summary**

* To **disable subnets** → just pass **empty lists** ([]) for public\_subnets, private\_subnets, or database\_subnets.
* To **disable NAT Gateway** → set enable\_nat\_gateway = false.
* To **create only VPC** → use empty lists for all subnets.

⚡ Pro tip: The **terraform-aws-modules/vpc/aws** module has tons of toggles like enable\_vpn\_gateway, enable\_flow\_log, enable\_dns\_hostnames, etc. You can selectively **turn features ON/OFF** instead of creating everything by default.

Awesome—let’s design a clean, scalable **Terraform architecture** that uses **modules + workspaces** to drive **multiple environments** (dev / stage / prod), and give you ready-to-run scripts & pipeline examples.

**🧱 Architecture Overview**

**Goal:** One codebase, reusable **modules**, isolated **states** per environment via **workspaces**, and simple automation.

**Recommended repo layout (two-repo pattern):**

* **infra-modules/** → pure reusable modules (versioned, no state here)
* **infra-live/** → stacks using modules + workspaces (holds backend, providers, env config)

**Logical design**

* **State:** Remote backend (S3) with DynamoDB lock, **1 state per workspace**.
* **Envs:** dev, stage, prod as **Terraform workspaces**.
* **Config:** Small \*.tfvars per workspace or locals map.
* **Modules:** network (VPC), eks (or ecs), rds, s3, etc.
* **Identity:** (Optional) multi-account: assume role per environment.

┌──────────────────────────┐

│ infra-modules (library) │

│ vpc, eks, rds, s3,... │

└───────────┬──────────────┘

│ (versioned "source")

┌───────▼──────────────────┐

│ infra-live (stacks) │

│ backend + providers │

│ root modules use source │

└───┬──────────┬───────────┘

│ │

Workspaces: dev stage prod (isolated states)

**📁 Folder Structure (infra-live)**

infra-live/

├─ backend.tf

├─ providers.tf

├─ variables.tf

├─ versions.tf

├─ main.tf

├─ outputs.tf

├─ env/

│ ├─ dev.tfvars

│ ├─ stage.tfvars

│ └─ prod.tfvars

└─ scripts/

├─ tfw.sh

└─ bootstrap\_backend\_once.tf (or separate “bootstrap” folder)

**infra-modules/** (example)

infra-modules/

├─ vpc/

│ ├─ main.tf

│ ├─ variables.tf

│ └─ outputs.tf

├─ eks/

│ ├─ main.tf

│ ├─ variables.tf

│ └─ outputs.tf

└─ rds/

├─ main.tf

├─ variables.tf

└─ outputs.tf

**☁️ Remote State & Locking**

**backend.tf**

terraform {

backend "s3" {

bucket = "mycompany-tfstate" # create once

dynamodb\_table = "mycompany-tflock" # create once

region = "ap-south-1"

encrypt = true

key = "global/${terraform.workspace}/infra.tfstate"

}

}

key embeds the workspace → each workspace gets its own state object.

Bootstrap the S3 bucket + DynamoDB **once** (by hand or a tiny one-off TF stack).

**👤 Providers & (Optional) Multi-Account AssumeRole**

**providers.tf**

provider "aws" {

region = var.region

# OPTIONAL (multi-account):

assume\_role {

role\_arn = local.env\_to\_role[terraform.workspace]

session\_name = "tf-${terraform.workspace}"

}

}

locals {

env\_to\_role = {

dev = "arn:aws:iam::111111111111:role/TerraformExecutionRole"

stage = "arn:aws:iam::222222222222:role/TerraformExecutionRole"

prod = "arn:aws:iam::333333333333:role/TerraformExecutionRole"

}

}

variable "region" {

type = string

default = "ap-south-1"

}

**🧩 Root Module Wiring (uses modules)**

**main.tf**

locals {

# Minimal per-env knobs; keep sensitive values in SSM/Secrets Manager.

env = terraform.workspace

cidr\_map = {

dev = "10.0.0.0/16"

stage = "10.1.0.0/16"

prod = "10.2.0.0/16"

}

eks\_node\_count = {

dev = 2

stage = 3

prod = 6

}

db\_instance\_class = {

dev = "db.t4g.micro"

stage = "db.t4g.small"

prod = "db.m6g.large"

}

tags = {

Project = "myapp"

Environment = local.env

ManagedBy = "Terraform"

}

}

module "vpc" {

source = "git::https://github.com/myorg/infra-modules.git//vpc?ref=v1.2.0"

name = "myapp-${local.env}-vpc"

cidr = local.cidr\_map[local.env]

az\_count = 3

tags = local.tags

}

module "eks" {

source = "git::https://github.com/myorg/infra-modules.git//eks?ref=v1.2.0"

name = "myapp-${local.env}"

vpc\_id = module.vpc.vpc\_id

private\_subnets = module.vpc.private\_subnet\_ids

desired\_size = local.eks\_node\_count[local.env]

min\_size = 1

max\_size = 10

tags = local.tags

}

module "rds" {

source = "git::https://github.com/myorg/infra-modules.git//rds?ref=v1.2.0"

name = "myapp-${local.env}"

engine = "postgres"

engine\_version = "15"

class = local.db\_instance\_class[local.env]

vpc\_id = module.vpc.vpc\_id

subnets = module.vpc.private\_subnet\_ids

allow\_from\_sg\_ids = [module.eks.node\_sg\_id]

multi\_az = local.env == "prod" ? true : false

deletion\_protection = local.env == "prod"

tags = local.tags

}

**outputs.tf**

output "vpc\_id" { value = module.vpc.vpc\_id }

output "eks\_cluster" { value = module.eks.cluster\_name }

output "db\_endpoint" { value = module.rds.endpoint }

**🧰 Example Module Stubs (infra-modules)**

**vpc/main.tf**

variable "name" { type = string }

variable "cidr" { type = string }

variable "az\_count" { type = number }

variable "tags" { type = map(string) }

# ... create VPC, subnets, IGW, NAT, route tables

# expose outputs: vpc\_id, private\_subnet\_ids, public\_subnet\_ids

output "vpc\_id" { value = aws\_vpc.this.id }

output "private\_subnet\_ids" { value = aws\_subnet.private[\*].id }

output "public\_subnet\_ids" { value = aws\_subnet.public[\*].id }

**eks/main.tf** & **rds/main.tf** follow the same pattern: accept inputs, create resources, expose outputs.

**🧪 Per-Environment Variables (optional)**

If you prefer **tfvars files** over locals maps:

**env/dev.tfvars**

region = "ap-south-1"

eks\_desired\_size = 2

db\_instance\_class = "db.t4g.micro"

tags = {

CostCenter = "1001"

}

…and similarly for stage.tfvars, prod.tfvars.  
Wire them through variables in variables.tf, and pass via CLI (script below).

**🚀 Workspace Script (create, select, plan, apply)**

**scripts/tfw.sh**

#!/usr/bin/env bash

set -euo pipefail

CMD="${1:-plan}" # plan | apply | destroy | output

ENV="${2:-dev}" # dev | stage | prod

TFVARS="env/${ENV}.tfvars"

terraform init -upgrade

# Create workspace if missing

if ! terraform workspace list | sed 's/\*//g' | tr -d ' ' | grep -q "^${ENV}$"; then

terraform workspace new "${ENV}"

fi

terraform workspace select "${ENV}"

case "$CMD" in

plan)

terraform plan -var-file="${TFVARS}"

;;

apply)

terraform apply -var-file="${TFVARS}" -auto-approve

;;

destroy)

terraform destroy -var-file="${TFVARS}" -auto-approve

;;

output)

terraform output -json

;;

\*)

echo "Usage: $0 {plan|apply|destroy|output} {dev|stage|prod}"

exit 1

;;

esac

**Examples**

./scripts/tfw.sh plan dev

./scripts/tfw.sh apply stage

./scripts/tfw.sh destroy prod

**🤖 CI/CD Pipeline (Jenkins & GitHub Actions)**

**Jenkins (Declarative)**

pipeline {

agent any

environment {

AWS\_DEFAULT\_REGION = 'ap-south-1'

}

stages {

stage('Checkout') {

steps { checkout scm }

}

stage('Terraform Init') {

steps { sh 'terraform init -upgrade' }

}

stage('Plan Matrix') {

matrix {

axes {

axis { name 'ENV'; values 'dev','stage','prod' }

}

stages {

stage('Select Workspace & Plan') {

steps {

sh """

./scripts/tfw.sh plan ${ENV}

"""

}

}

}

}

}

stage('Apply Prod (manual)') {

when { expression { return params.APPLY\_PROD == true } }

steps { sh './scripts/tfw.sh apply prod' }

}

}

parameters {

booleanParam(name: 'APPLY\_PROD', defaultValue: false, description: 'Apply to prod')

}

}

**GitHub Actions (Matrix)**

name: Terraform

on:

workflow\_dispatch:

inputs:

action:

description: plan/apply/destroy

required: true

default: plan

jobs:

tf:

runs-on: ubuntu-latest

strategy:

matrix:

env: [dev, stage, prod]

steps:

- uses: actions/checkout@v4

- uses: hashicorp/setup-terraform@v3

- name: Configure AWS creds

uses: aws-actions/configure-aws-credentials@v4

with:

role-to-assume: ${{ secrets.ROLE\_ARN }}

aws-region: ap-south-1

- name: Terraform ${{ github.event.inputs.action }} (${{ matrix.env }})

run: |

chmod +x scripts/tfw.sh

./scripts/tfw.sh "${{ github.event.inputs.action }}" "${{ matrix.env }}"

**🛡️ Guardrails & Best Practices**

* **State isolation:** embed terraform.workspace in backend key.
* **IAM boundaries:** (Multi-account) map workspace → role ARN; least-privilege execution role.
* **Tagging:** enforce common tags in modules; add env tag from workspace.
* **Production safety:** deletion\_protection = terraform.workspace == "prod".
* **Data separation:** avoid cross-workspace data sources; never reference one env’s state from another.
* **Secrets:** store in **AWS Secrets Manager / SSM Parameter Store**; read via data sources.
* **Module versions:** pin ?ref=vX.Y.Z to avoid surprise upgrades.
* **Plan gates:** require manual approval for prod apply.
* **Cost controls:** smaller sizes in dev/stage; autoscaling caps per env.

**🧪 Quickstart (local)**

terraform init

terraform workspace new dev

terraform workspace new stage

terraform workspace new prod

./scripts/tfw.sh plan dev

./scripts/tfw.sh apply dev