Devops Shack

Terraform Modules & Reusability

By Devops Shack

DevOps Shack

Terraform Modules & Reusability

Table of Contents

1. Reusable Terraform Modules: The Right Way

Reusable modules are the backbone of scalable, maintainable Infrastructure as Code. They let you package and reuse logic like VPC creation, EC2 provisioning, IAM roles, Kubernetes clusters, etc., across multiple environments and teams
— without repeating yourself.

1.1 Why Modules Matter

Why not just write everything in one big .tf file?

Because as infrastructure grows:

- You need **reusability** across environments (dev/stage/prod).
- You need **separation of concerns** (networking vs compute vs DNS).
- You need **versioning** of components.
- You want **team-level ownership** of modules.

Benefits:

- DRY code (Don't Repeat Yourself)
- Easier testing and iteration
- Portability (across clouds/teams)
- Easier collaboration in Git

🧠 *Think of modules as Terraform's version of a function or class.*

1.2 Creating a Basic Module Structure

Let's say we want to build a reusable **VPC module**.

📁 Directory layout:

```
project/
├── main.t                f     # Root configuration
└── modules/
    └── vpc/
```

```
├── main.tf# Resource definitions
    ├── variables.t    f# Inputs with defaults
    └── outputs.t      f    # Outputs to expose
```

modules/vpc/main.tf

```
resource "aws_vpc" "this"
{ cidr_block = var.cidr_block
tags = {
   Name = var.name
 }
}
```

modules/vpc/variables.tf

```
variable "cidr_block"
 { type     = string
 description = "CIDR block for the VPC"
}

variable "name"
 { type     = string
 description = "Name tag for the VPC"
}
```

modules/vpc/outputs.tf

```
output "vpc_id" {
 value      = aws_vpc.this.id
 description = "The ID of the created VPC"
}
```

1.3 Module Calling Example

Now let's call this module in our **root** Terraform configuration:

main.tf

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

module "vpc" {
 source     = "./modules/vpc"
 name       = "dev-vpc"
 cidr_block = "10.0.0.0/16"
}

output "vpc_id" {
 value = module.vpc.vpc_id
}
```

Result:

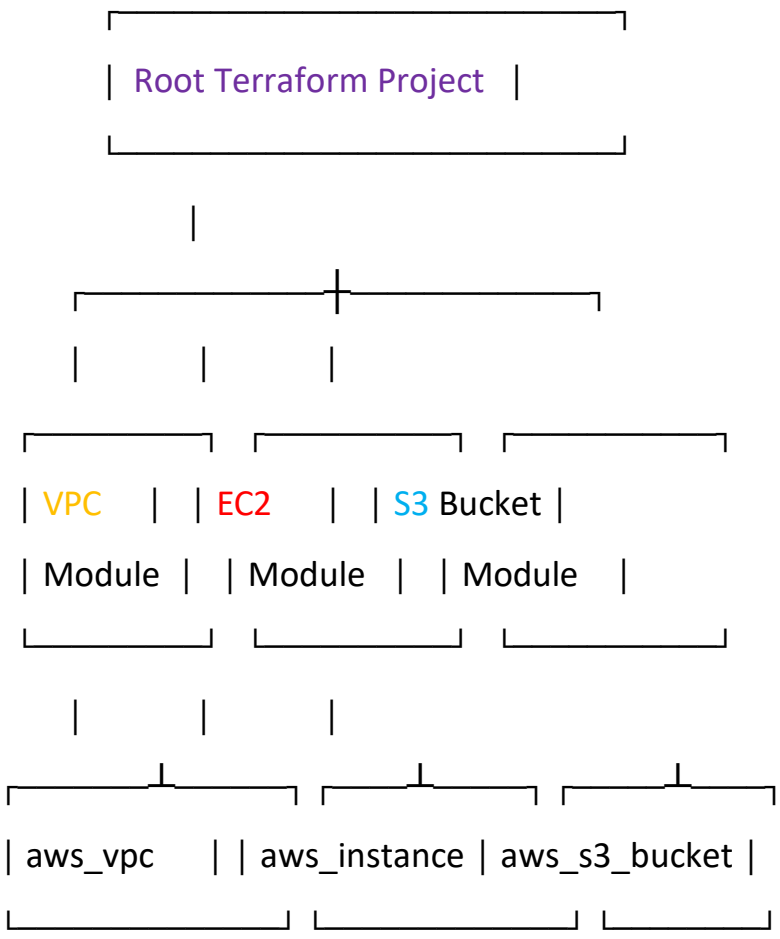
- Terraform will create a VPC with the provided CIDR and name.
- You can reuse this module in prod/, stage/, etc., just by changing variables.

Real-World Use Cases for Modules

Resource	Module Name	Example Usage
VPC	vpc	module "vpc" { source = "./modules/vpc" }
EC2 Instances	ec2_instance	Pass AMI, instance type, key name

Resource	Module Name	Example Usage
IAM Roles/Policy	iam_roles	Modularized permission assignments
EKS Cluster	eks_cluster	Includes cluster, node groups, IRSA, etc.
S3 Buckets	s3_bucket	For logging, backups, lifecycle policies

Visual Diagram (Conceptual)



Each of these modules can be developed, versioned, and tested **independently** — and reused across many projects.

Tips for Robust Modules

- Use variable types and default values to make inputs safe.
- Expose only necessary outputs.

Add description fields to help others use your module.

- Always version modules using Git tags or semantic versioning.
- Add an example in a /examples folder for clarity.

🏛 2. Creating a Scalable Module Registry

A **module registry** is the foundation of organized, consistent, and production- grade Terraform usage across teams and environments. It allows you to share reusable modules either:

- Locally (within your project or org)
- Via Git repositories
- Or with a centralized **Terraform Registry** (private or

public) Let's explore how to build and manage one the right

way.

2.1 Organizing Module Repos

There are two popular patterns for organizing modules:

☑ **Option 1: Mono-Repo Structure**

terraform-modules/

├── vpc/

├── ec2/

├── rds/

├── alb/

Pros:

- Easy versioning of all modules together
- Common tooling (linting, tests) is simpler

Cons:

- Tagging or promoting a single module requires full repo versioning

☑ **Option 2: Poly-Repo Structure**

Each module lives in its own Git

repository: terraform-vpc/

terraform-ec2/

terraform-alb/

Pros:

Fine-grained versioning

- Easier ownership by different teams

Cons:

- Requires more repo management and CI/CD setup

📌 **Best Practice:**
If your team is small, start with a **mono-repo**. As you scale, adopt **poly-repo** with clear naming and CI automation.

2.2 Publishing to Terraform Registry (Private or Public)

You can make your modules easily consumable via:

▨ **Public Terraform Registry**

To publish:

- Host your module in GitHub
- Follow naming conventions:
  - Repo: terraform-<PROVIDER>-<NAME> → terraform-aws-vpc
  - Inside the repo: module source at root (main.tf, variables.tf,

etc.) Then it can be consumed like this:

```
module "vpc" {
 source = "terraform-aws-modules/vpc/aws"
 version = "3.0.0"
}
```

▨ **Private Terraform Registry**

Terraform Cloud (TFC) or Terraform Enterprise supports private registries.

Steps:

1. Upload your module via UI or CLI
2. Use like this:

```
module "vpc" {
source = "app.terraform.io/org/vpc/aws" version =
 "1.0.2"
}
```

📌 This enables fine-grained access control, audit logging, and policy enforcement.

2.3 Documentation and Usability

Well-documented modules are **10x more usable**.

Here's what to include:

☑ **README.md**

Explain:

- Module purpose
- Required and optional inputs
- Outputs
- Examples

☑ **Auto-generate Docs with terraform-docs**

Install:

`brew install terraform-docs` Run:

`terraform-docs markdown table ./modules/vpc > README.md`

It extracts inputs/outputs automatically and formats them into Markdown.

☑ **Add Example Usage**

🗀 Directory structure:

```
modules/
└── vpc/
    └── examples/
```

└── basic/

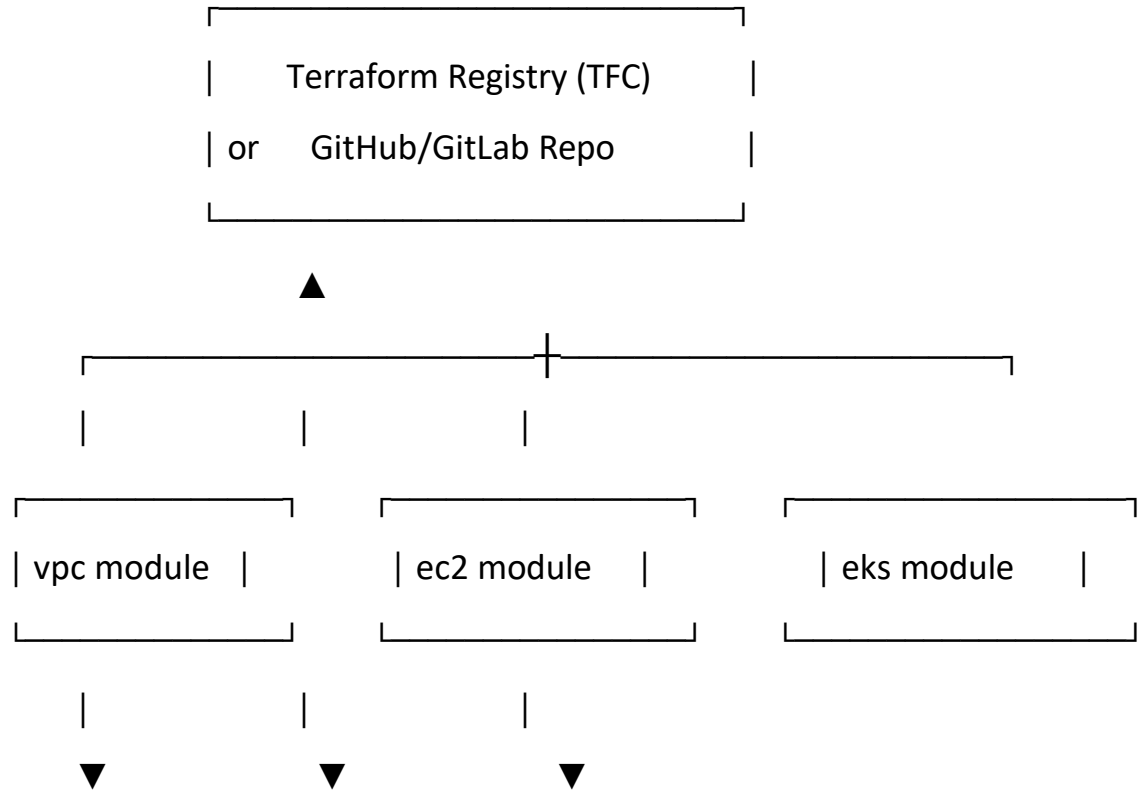
├── main.tf

├── terraform.tfvars

These examples:

- Show how to use the module
- Can be tested with Terratest or CI
- Help future contributors or consumers

Conceptual Diagram: Module Registry



Summary: Scalable Module Registry

Element	Key Considerations
Structure	Choose mono- or poly-repo wisely

Element	Key Considerations
Versioning	Use semantic Git tags
Registry	Publish to public or private registry
Docs	Use terraform-docs and README best practices
Examples	Always include basic usage examples

🔄 3. When to Use for_each vs count in Terraform Modules

One of the most common questions in Terraform module development is:
“Should I use count or for_each?”

While both allow **resource duplication**, their usage patterns, flexibility, and readability differ significantly — especially inside reusable modules.

Let's explore the differences in depth.

3.1 count – Simple Replication

count is the simplest and oldest way to create multiple resources of the same type.

☑ **Example: Creating 3 EC2**

Instances resource "aws_instance"

"web" { count = 3

ami = var.ami

instance_type = var.instance_type

tags = {

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

}

}

- Creates 3 instances
- count.index returns 0, 1, 2 (useful for naming)
- Resource addresses become:
  - aws_instance.web[0]
  - aws_instance.web[1]
  - aws_instance.web[2]

⚠ **Drawbacks of count**

- Index-based → no named identification
- If you delete an item from the middle, **Terraform re-creates** all resources after it

Complex when looping over maps/sets

- Not ideal for **selective targeting** in large projects

3.2 for_each – Named and Indexed Objects

for_each is more expressive, cleaner, and safer for most real-world use cases.

☑ **Example: Create EC2 Instances from Map**

```
variable "instances"
{ type =
map(object({ ami =
string
type = string
}))
}

resource "aws_instance" "web"
{ for_each = var.instances
ami = each.value.ami
instance_type = each.value.type
tags = {
Name = each.key
}
}
```

🔍 Given this input:

```
instances =
{ "frontend" = {
ami = "ami-123" type
= "t2.micro"
```

```
  },
  "backend" = {
    ami  = "ami-456"
    type = "t2.small"
  }
}
```

Terraform creates:

- aws_instance.web["frontend"]
- aws_instance.web["backend"]

💡 These resource addresses are **human-readable** and **stable**, even if others are removed or added.

🧠 Benefits of for_each

Feature	count	for_each
Indexing	Integer	Named key
Loop over map/set	❌ Not ideal	☑ Native
Stable references	❌ No (fragile)	☑ Yes
Selective deletion	❌ Tricky	☑ Easy (terraform taint/destroy)
Input data type	List or integer	Map or set

3.3 When Used in Modules

You can use both count and for_each **inside modules** or even to **instantiate modules** multiple times.

☑ Using for_each to Instantiate a Module Dynamically

```
module "s3_bucket" {
```

```
 for_each = {
  logs    = "log-bucket"
  archive = "archive-bucket"
 }

 source =
 "./modules/s3" name  =
 each.value  tags = {
  Purpose = each.key
 }
}
```

- Will create two S3 buckets using the same module
- name is passed from the values
- tags.Purpose uses the key ("logs" or "archive")

☑ **Inside a Module: Use for_each Over Input Map**

```
resource "aws_security_group_rule" "rules"
{ for_each = var.ingress_rules
 type     = "ingress"
 from_port = each.value.from
 to_port  = each.value.to
 protocol = each.value.protocol
 cidr_blocks = each.value.cidr
}
```

Input:

hcl

```
ingress_rules =
 { "ssh" = {
   from     = 22
   to       = 22
   protocol =
   "tcp"
   cidr     = ["0.0.0.0/0"]
 }
}
```

Conceptual Use Decision Flow

Are you looping over a list? ——▶ Use `count`

Are you looping over a map/set with keys? ——▶ Use `for_each`

Do you need stable, named resources? ——▶ Use `for_each`

Do you want index-based resources only? ——▶ Use `count`

🔧 **Pro Tips for Real-World Use**

- Use for_each for anything dynamic, especially in modules
- Avoid using both count and for_each together on the same resource
- For lists of complex objects, consider converting to a map using zipmap for for_each compatibility
- Avoid overengineering — use the simpler option unless dynamic behavior is required

4. Module Versioning and Upgrades Without Downtime

Versioning Terraform modules is **crucial** in large-scale environments where infrastructure changes need to be predictable, auditable, and **non-disruptive**. This section teaches you how to handle module upgrades with **minimal risk** and **maximum control**.

4.1 Version Control with Git Tags

Terraform modules are often stored in **Git repositories**, and versioning is best done using **semantic Git tags** like:

v1.0.0 → First stable release

v1.1.0 → Minor feature addition

v2.0.0 → Breaking changes

☑ **To tag your module:**

```
git tag v1.0.0
git push origin v1.0.0
```

Now in your consuming Terraform config, pin that version:

```
module "vpc" {
  source = "git::https://github.com/org/terraform-vpc.git?ref=v1.0.0"
  name   = "prod-vpc"
  cidr_block = "10.0.0.0/16"
}
```

🔒 **Pin your module versions** instead of using main or master — this avoids unintentional upgrades that may break infra.

4.2 Upgrade Strategy

Suppose a module evolves from v1.0.0 → v1.1.0 by adding an optional output.

Here's how to safely upgrade the module **without breaking downstream infrastructure**.

🔄 **Step-by-step Upgrade:**

1. **Update Module Version**:

```
source  = "git::https://github.com/org/terraform-vpc.git?ref=v1.1.0"
```

2. **Run Plan**:

```
terraform get -
update
```

```
terraform plan
```

This tells Terraform to re-fetch the module source.

3. **Validate Output Changes**
   Ensure nothing in the module causes resource destruction (like changes to name, tags, etc.).
4. **Apply** if plan looks

safe: terraform apply

📌 For major updates (v1 → v2), always read changelogs, or test module upgrades in **lower environments** first.

🚫 Avoid these risky practices:

- Pulling directly from main branch
- Making breaking changes without version bumps
- Mutating outputs or inputs in-place without migration strategy

4.3 Blue-Green or Canary Module Rollouts

In production systems, you may want to **gradually apply module upgrades** to reduce risk.

Let's look at two advanced rollout patterns.

☑ A. Blue-Green Infra Module Strategy

Maintain two versions of infra (e.g., VPC or cluster):

```
module "vpc_blue" {
 source  = "git::https://github.com/org/terraform-vpc.git?ref=v1.0.0"
   ...
}
```

```
 module "vpc_green" {
 source  = "git::https://github.com/org/terraform-vpc.git?ref=v1.1.0"
 ...
}
```

Deploy both → test vpc_green → switch traffic via Route 53 or ALB → destroy vpc_blue.

This is useful when changes involve high risk (like subnet layout or DNS zones).

☑ B. Canary Rollouts in Module Consumers

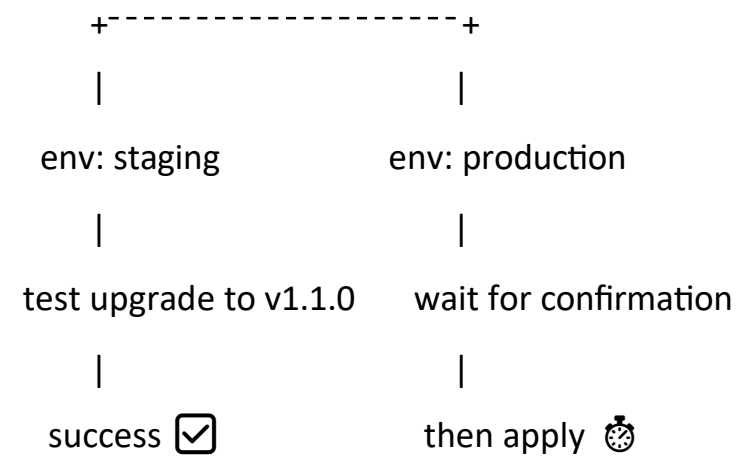
Update just **one environment or region**:

```
module "vpc_ap-south-1" {
 source  = "git::https://github.com/org/terraform-vpc.git?ref=v1.1.0"
}
module "vpc_us-east-1" {
 source  = "git::https://github.com/org/terraform-vpc.git?ref=v1.0.0"
}
```

Test success in ap-south-1 before upgrading us-east-1.

Visual Rollout Flow (Conceptual)

```
        +--------------+
        |  Module v1.0.0  |
        +--------------+
              |
```



🔧 Bonus: Module Registry Versioning

If you're using **Terraform Cloud/Enterprise** or the **Public Registry**, you can release tagged versions with documentation, and users can consume like this:

```
module "vpc" {
  source = "app.terraform.io/org/vpc/aws"
  version = "1.2.3"
}
```

☑ This allows:

- Locked module versions
- Easy rollback
- History and changelog tracking

📌 Summary: Versioning & Upgrades

Best Practice	Why It Matters
Use semantic Git tags	Clear upgrade paths, rollback possible
Never reference main in prod	Prevents accidental breaking changes
Upgrade gradually with canary/blue-green	Reduce risk in critical environments

Best Practice	Why It Matters
Validate terraform plan first	Catch unintended resource replacement
Document module changes	Enable trust, visibility, and reusability

🧱 5. Terraform Composition: Layered Design Patterns

As infrastructure grows in complexity, we must **compose modules in a clean, layered, and reusable fashion**. Terraform composition enables teams to build logical stacks, reuse modules intelligently, and maintain clarity between layers (networking, compute, services, etc.).

Let's look at how to do it the **right way**.

5.1 Layered Module Structure

Layered composition means designing infrastructure as **logical tiers**, such as:

1. Base Infrastructure Layer → VPC, Subnets, IAM, KMS
2. Compute Layer → EC2, ECS, EKS, Lambda
3. Services Layer → Databases, Caches, Queues
4. Platform Layer → DNS, Monitoring, TLS, Vault
5. Application Deployment Layer → App-specific resources

📁 Suggested structure:

```
terraform/
├── layers/
│   ├── networking/
│   ├── compute/
│   ├── services/
│   └── application/
├── modules/
└── environments/
    ├── dev/
    └── prod/
```

Each layer **calls reusable modules** from the modules/ folder or external registries.

☑ **Example: Layered Root Config**

📄 terraform/layers/networking/main.tf

```
module "vpc" {
  source    =
  "../../modules/vpc" name
  = "dev-vpc" cidr_block =
```

"10.0.0.0/16"

```
}
```

📄 terraform/layers/compute/main.tf

```
module "eks" {
  source     = "../../modules/eks"
  cluster_name = "dev-cluster"
  vpc_id       = module.vpc.vpc_id
}
```

☑ Each layer is self-contained and focused on **a single responsibility**.

5.2 Shared Module Patterns

When building layers, several **module design patterns** help ensure DRY code and better scalability:

▧ A. Wrapper Modules

Wrap external modules with organization-specific defaults.

```
module "org_vpc" {
  source = "terraform-aws-modules/vpc/aws"
  name    = var.name
  cidr    = var.cidr
  enable_dns_hostnames =
  true tags = {
    Owner = "platform-team"
  }
}
```

This ensures all teams inherit org-wide tagging, naming, and behavior.

▨ B. Nested Modules

One module can call **multiple inner modules** to compose behavior.

```
module "network" {
  source = "./modules/network"
}

# Inside
/modules/network/main.tf: module
"vpc" {
  source = "../vpc"
}
module "subnets"
  { source =
  "../subnets"
}
```

☑ This lets you abstract large chunks into a single higher-level interface.

▨ C. Multi-Provider Modules

When supporting multi-cloud or multi-region deployments, inject providers from the outside.

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

module "s3_logging"
  { source  = "./modules/s3"
  providers = {
```

```
  aws = aws.us-east-1
}
name = "logs"
```

```
}
```

5.3 Real-World Example Structure

Let's take a **real-world team-level design** with Dev, Staging, and Prod environments:

📁 Example layout:

```
environments/
├── dev/
│   └── main.tf → calls `../layers/networking`, `../layers/compute`
├── staging/
├── prod/
layers/
├── networking/
├── compute/
├── services/
```

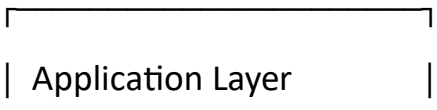
Each environment has:

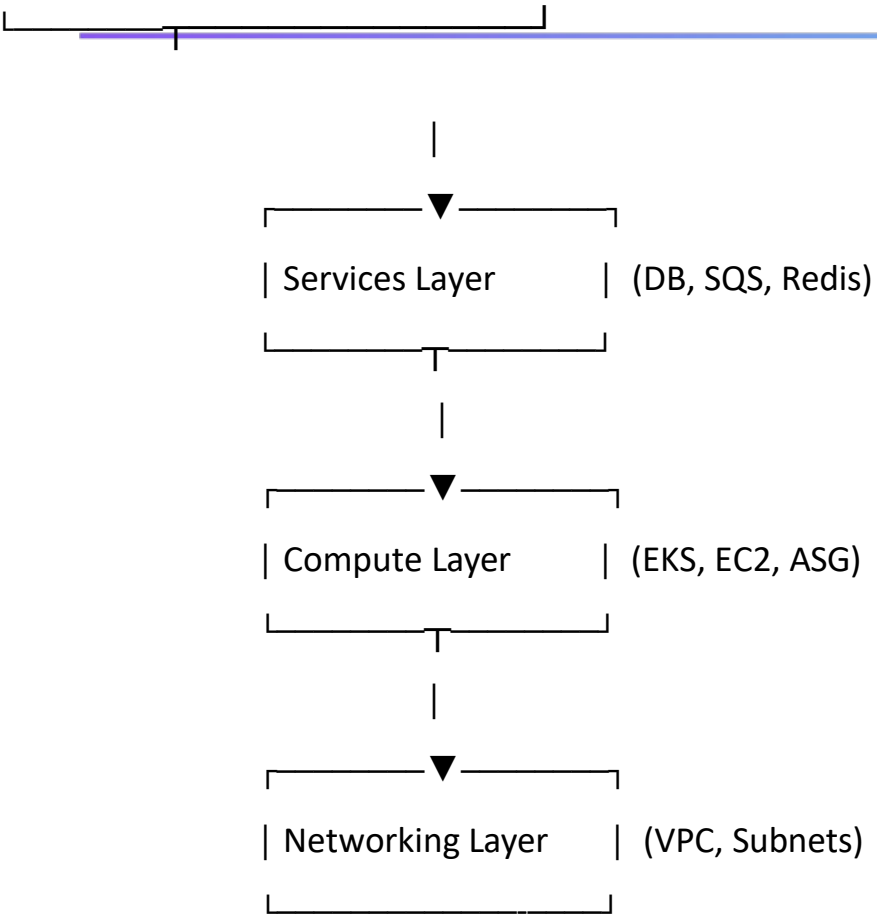
- **Isolated state files**
- **Backend config with remote state**
- **Composition of common modules across tiers**

☑ Benefits:

- Clean separation of concerns
- Clear ownership
- Reusability across lifecycle stages

Diagram: Terraform Composition Layers





Each layer builds upon the previous. Outputs are passed as inputs downstream.

🧠 Summary: Layered Design for Composability

Pattern	Purpose
Layered Folders	Logical breakdown of infra (VPC → App)
Wrapper Modules	Customize upstream modules with org defaults
Nested Modules	Abstract multiple modules into one
Per-Env Composition	Dev/Staging/Prod separation
Provider Injection	Enables multi-cloud/multi-region patterns

📓 Conclusion: The Power of Terraform Modules & Composable Infrastructure

In the evolving landscape of infrastructure as code, **modularity and reusability are not luxuries — they are necessities**. As organizations scale across teams, cloud providers, and environments, maintaining consistency, security, and efficiency becomes a herculean challenge. Terraform modules provide the scaffolding to tame this complexity.

Throughout this guide, we explored how to build, structure, and version reusable modules that serve not just as tools, but as foundations for **platform thinking** in modern DevOps. We began by understanding how to create reusable modules "the right way" — focusing on clean inputs, outputs, and encapsulation principles. Just like in traditional software engineering, clean boundaries and reusable components lead to scalable systems.

We then moved on to the idea of a **scalable module registry** — whether internal or external — and how it's the cornerstone of adoption within larger teams. Versioning modules using Git tags or Terraform Registry ensures that upgrades are predictable, changes are trackable, and rollbacks are safe — reducing the risk of breaking production.

The choice between for_each and count is subtle yet crucial. While both serve resource replication, their behavior in maps, sets, and dynamic modules varies. for_each offers clarity, predictability, and named resources, while count serves well in simpler, list-based repetition. Understanding their nuances empowers teams to write intent-revealing code, a hallmark of maturity.

When dealing with **module upgrades**, the ability to safely test, release, and roll forward (or back) without causing downtime is a powerful capability. Leveraging Git tags, canary rollouts, and blue-green deployments at the module level not only mitigates risk but also reflects operational excellence.

Finally, we explored **Terraform composition** — the art of designing layered infrastructure using modules. Layered designs enforce separation of concerns, encapsulate domain logic (networking, compute, databases, etc.), and foster reuse across environments (Dev, QA, Prod). Wrapper modules, nested structures, provider injection, and environment-aware composition patterns elevate your module usage from good to great.

🚀 Where Do We Go From Here?

- **Treat modules as products.** Every module should have documentation, examples, changelogs, and versions — just like software libraries.
- **Integrate testing frameworks** like terratest to assert your modules behave correctly under different configurations.
- **Build your own internal module registry or leverage Terraform Cloud** to distribute reusable building blocks securely.
- **Automate CI/CD for modules** just as you would for apps. Validate changes, enforce linting, test examples, and tag versions automatically.

Reusability is DevOps Maturity

Teams that invest in reusable Terraform modules find themselves accelerating delivery, reducing duplication, onboarding faster, and deploying with confidence. They shift from writing one-off .tf files to thinking in **platform patterns** and **interface contracts** — just like a seasoned software architect.

As Terraform continues to evolve, so too must our discipline in using it. Treat every module like a promise to your team: that it's tested, safe to use, and easy to extend. Because in the end, the more reusable your infrastructure is, the more **scalable**, **secure**, and **sane** your entire cloud ecosystem becomes.