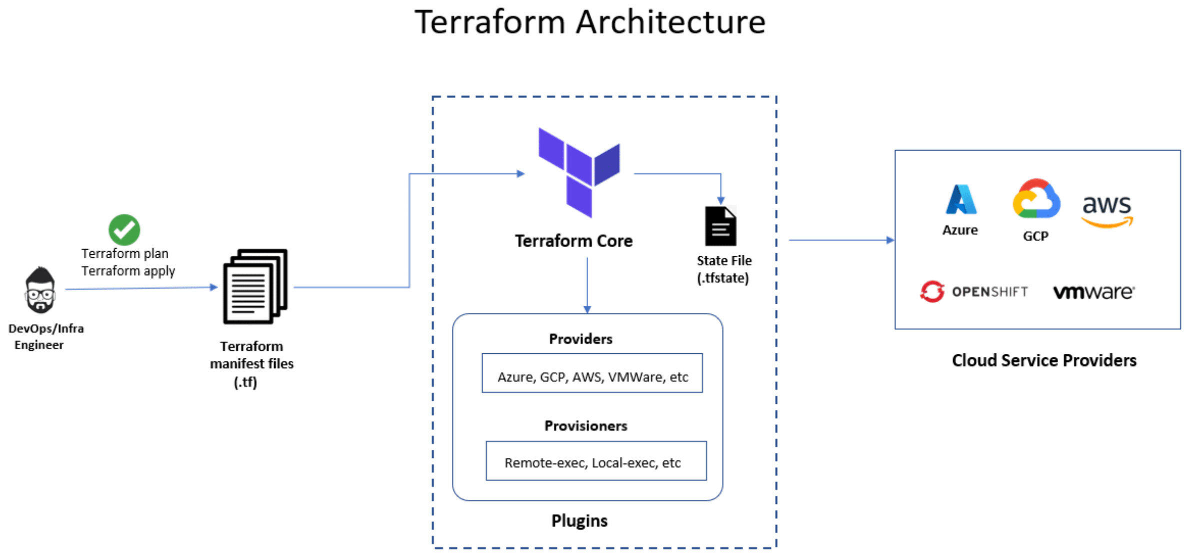
Module 4: Infrastructure as Code (IaC) – Terraform

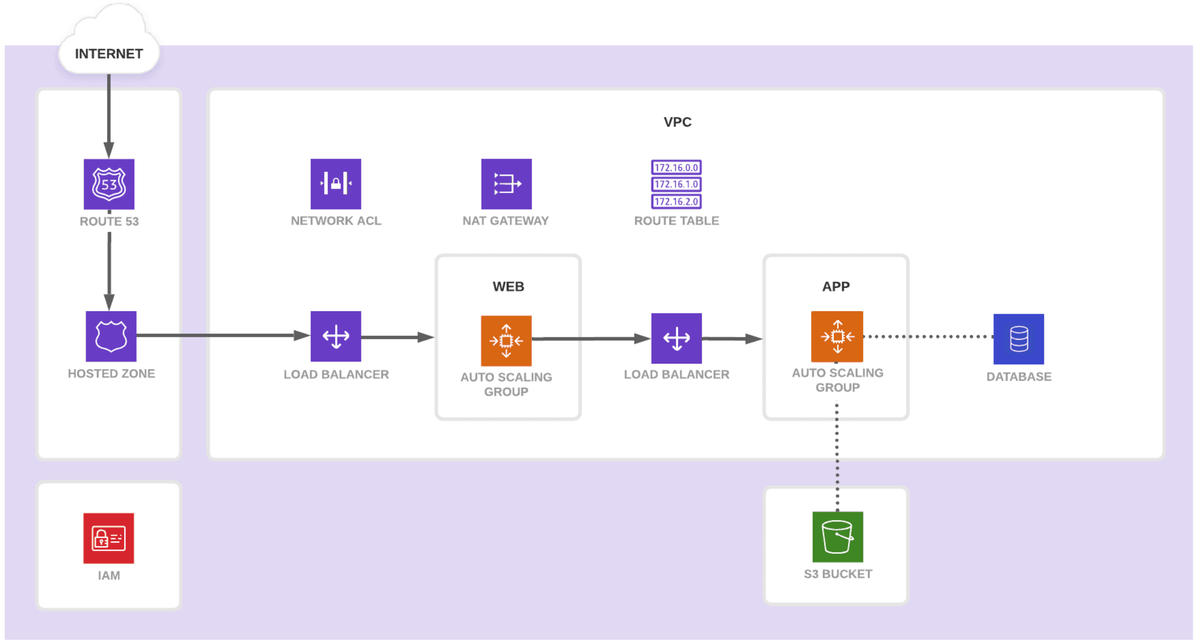
Topics Covered:

- Terraform modules and workspaces
- Remote state and state locking
- Terraform Cloud + VCS integration
- Multi-cloud deployments

---

1. Terraform Modules & Workspaces





1.1 What Are Terraform Modules?

A **module** is a group of Terraform configuration files **reusable** across projects.

Why Modules?

- Reusability
- Standardization
- Cleaner code structure
- Easier testing & versioning
- Strong DevOps automation

Module Structure

```
modules/
  vpc/
    main.tf
    variables.tf
    outputs.tf
  ec2/
    main.tf
    variables.tf
    outputs.tf
env/
```

```
  dev/
  prod/
```

Calling a Module

```
module "vpc" {
  source = "./modules/vpc"
  cidr_block = "10.0.0.0/16"
}
```

Public Module Sources

- Terraform Registry
- GitHub Repository
- Versioned modules:

```
source = "github.com/vivek/my-vpc-module?ref=v1.0.0"
```

1.2 Terraform Workspaces

Workspaces allow multiple **state files** for the same configuration.

Use Cases:

- dev, qa, prod environment switching
- Single module → multiple environments
- Avoid duplicated code

Commands

```
terraform workspace list
terraform workspace new dev
terraform workspace select prod
```

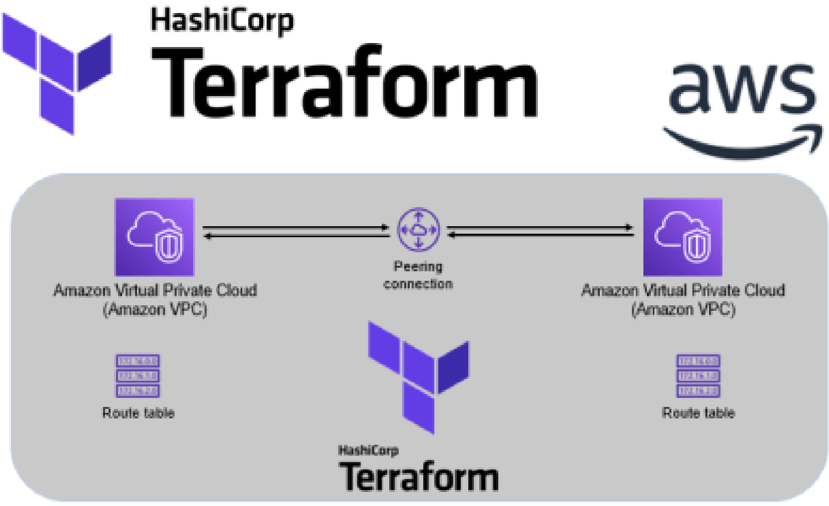
How Workspaces Work

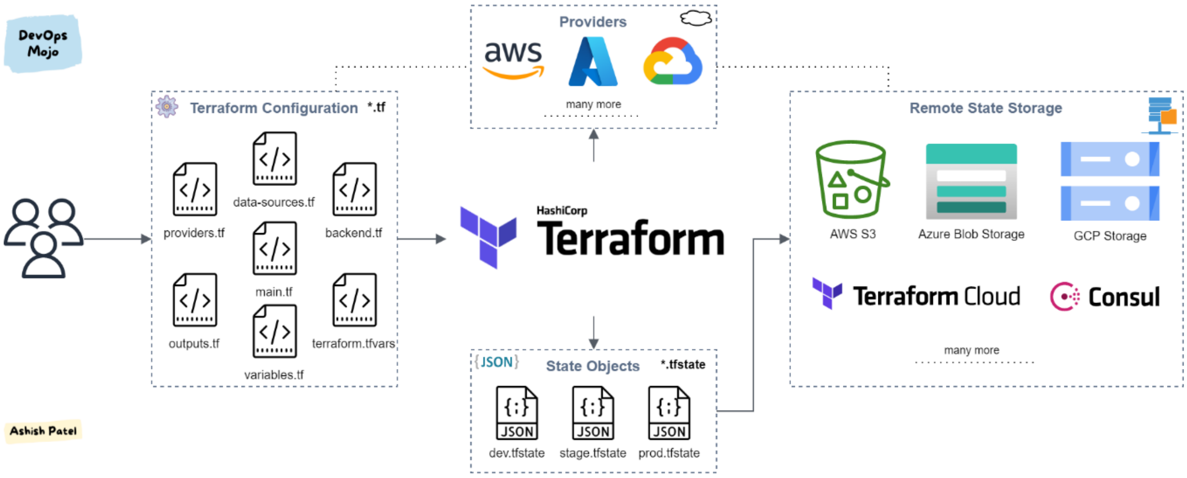
Each workspace creates its own state:

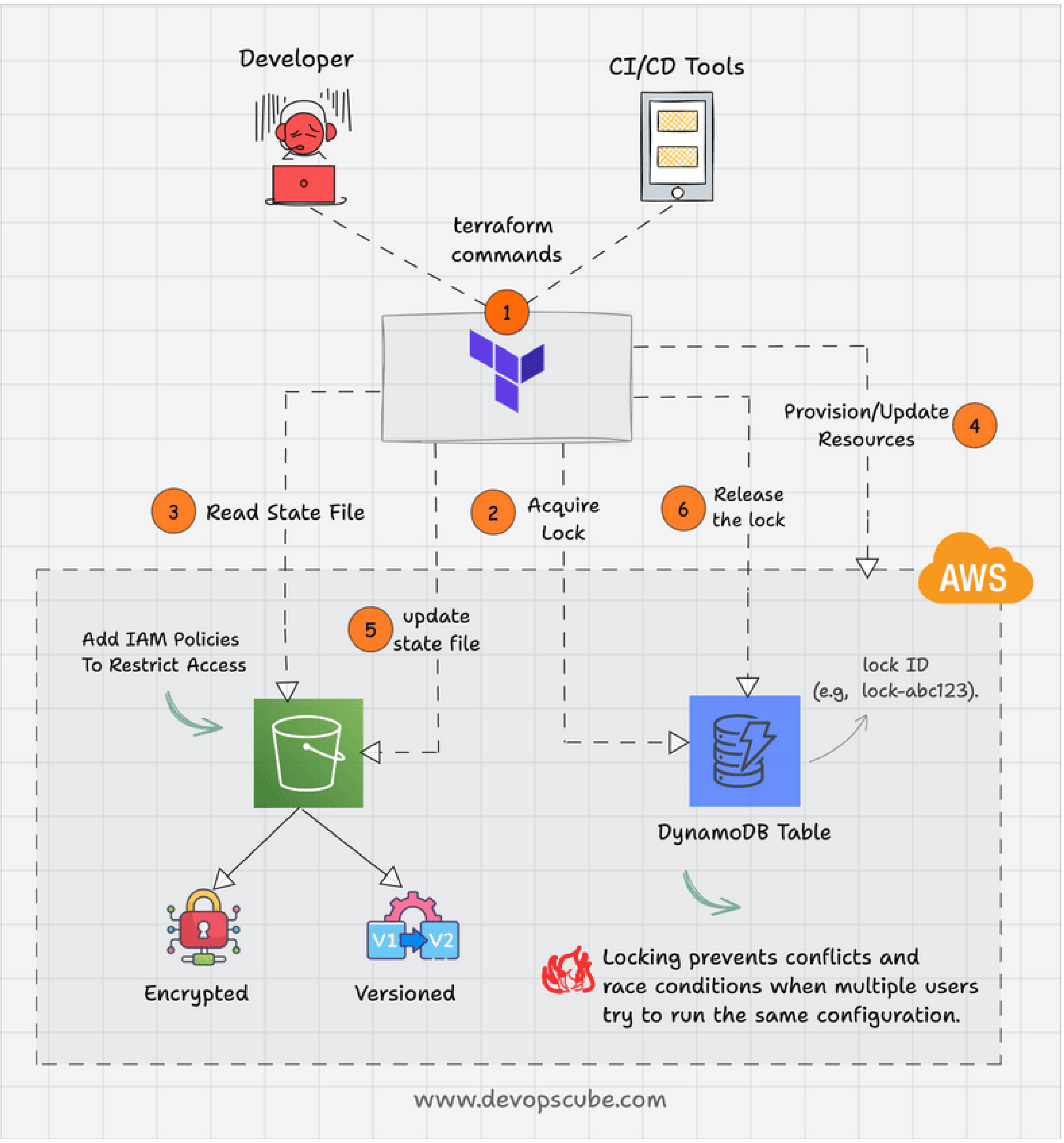
```
terraform.tfstate.d/dev/terraform.tfstate
terraform.tfstate.d/prod/terraform.tfstate
```

---

2. Remote State & State Locking


HashiCorp
Terraform
aws
Amazon Virtual Private Cloud (Amazon VPC)
Peering connection
Amazon Virtual Private Cloud (Amazon VPC)
Route table
Route table
HashiCorp
Terraform




Developer
CI/CD Tools
terraform commands
1
Provision/Update Resources
4
3
Read State File
2
Acquire Lock
6
Release the lock
AWS
Add IAM Policies To Restrict Access
5
update state file
lock ID (e.g, lock-abc123).
DynamoDB Table
Encrypted
Versioned
V1
V2
Locking prevents conflicts and race conditions when multiple users try to run the same configuration.
www.devopscube.com

Terraform uses state to keep track of the real infrastructure.
Remote state allows teams to share this state securely.

2.1 Why Remote State?

- Collaboration
- Centralization
- Security
- Automated pipelines

- Prevent state loss

2.2 Backends

Common Terraform backends:

Backend	State Locking	Recommended Use
S3 + DynamoDB	Yes	AWS production setups
Azure Storage	Yes	Azure IaC
Google Cloud Storage	Yes	GCP workloads
Terraform Cloud	Yes	Multi-cloud teams
Consul	Yes	HashiCorp heavy setups

2.3 Example: S3 Remote State + DynamoDB Locking

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

Benefits:

- Prevents multiple engineers from editing state
- Auto-lock/unlock on apply
- Secure, encrypted, versioned state

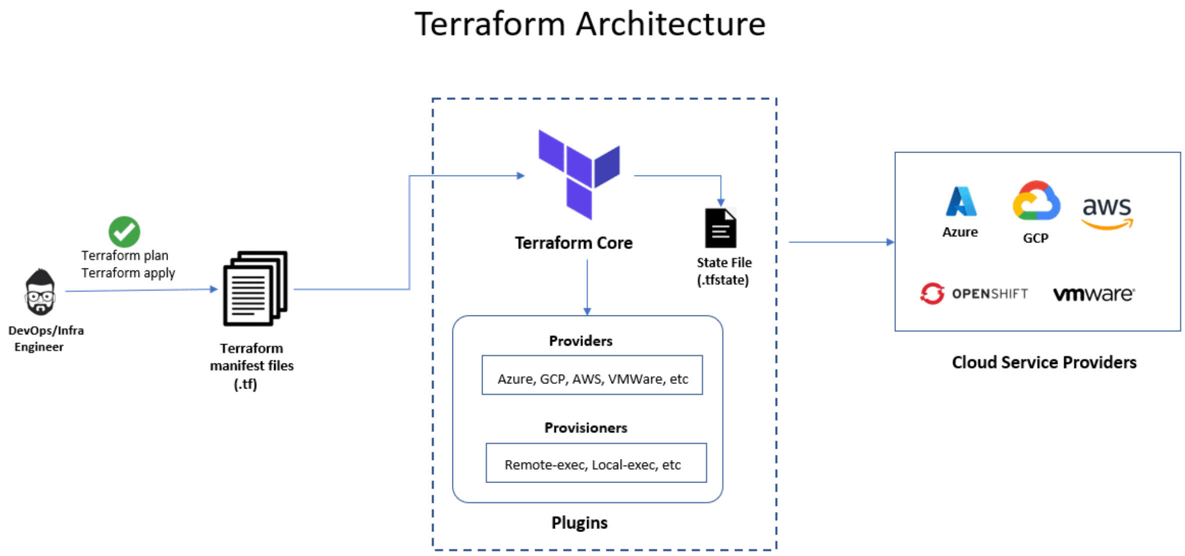
2.4 Remote State Data Source

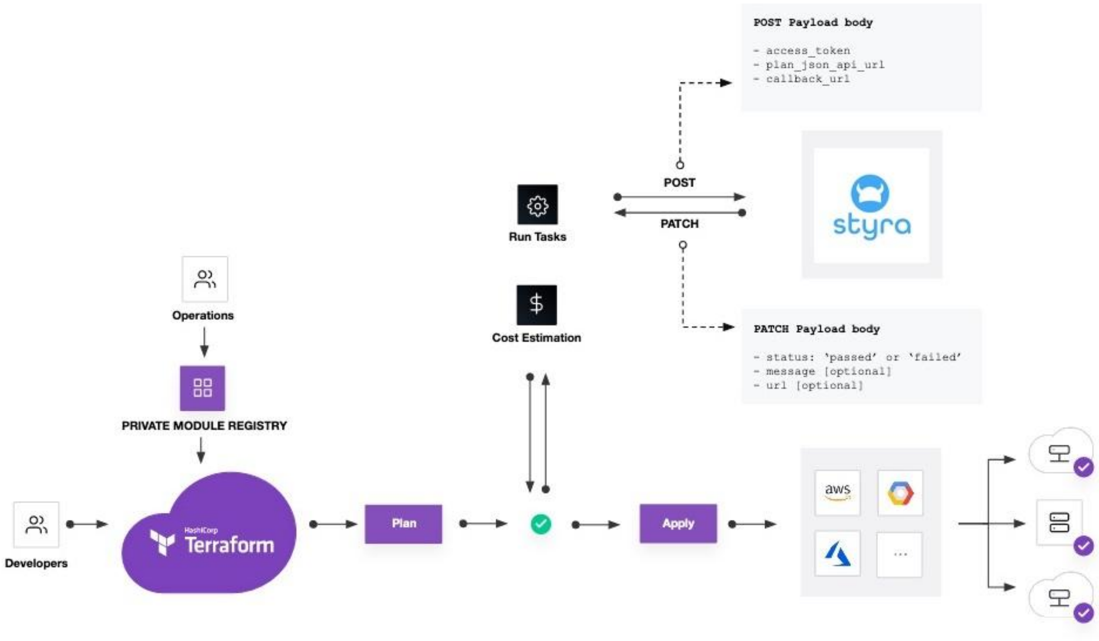
You can consume remote state from other projects:

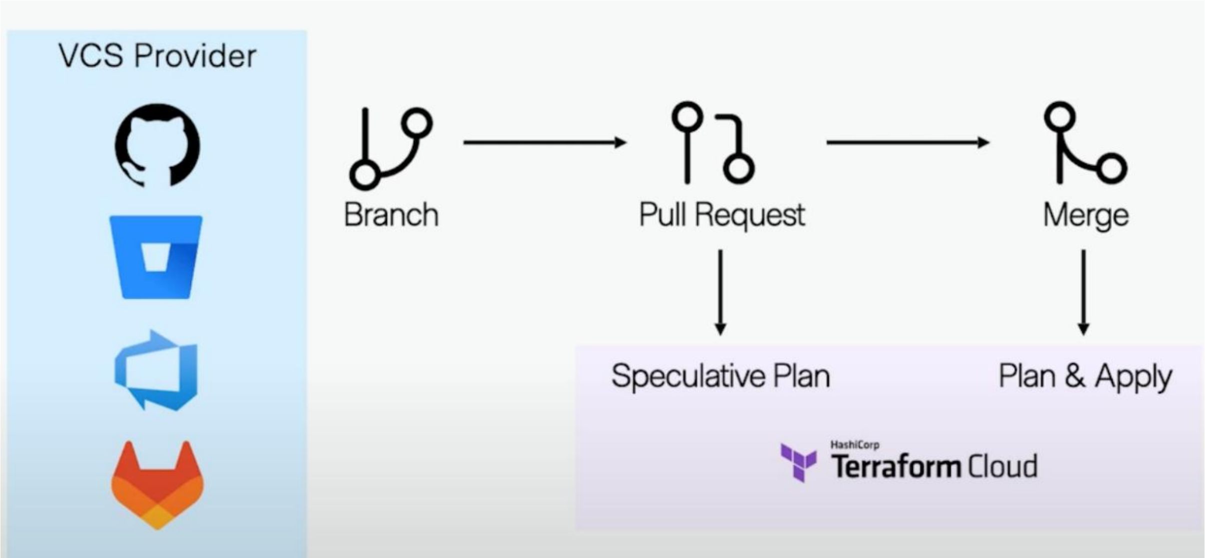
```
data "terraform_remote_state" "vpc" {
  backend = "s3"
  config = {
    bucket = "my-terraform-state"
    key    = "vpc/terraform.tfstate"
    region = "us-east-1"
  }
}
```

---

3. Terraform Cloud & Version Control Integration







Terraform Cloud (TFC) provides:

- Remote execution
- Version control integration
- Private module registry
- Policy-as-code with Sentinel
- State storage + locking
- Cost estimation

3.1 Connecting Terraform Cloud to GitHub/GitLab/Bitbucket

Steps:

1. Create a **Terraform Cloud Organization**
2. Add a **Workspace**
3. Connect VCS (GitHub, GitLab, Azure DevOps, Bitbucket)
4. Select the repository
5. Each git push triggers:
   - Plan
   - Cost analysis
   - Apply (manual or auto)

3.2 TFC Workspace Modes

Mode	Description
UI/VCS	Run triggered by Git
API-driven	Used from pipelines
CLI-driven	Local CLI with remote state

3.3 Example .terraformrc for Terraform Cloud

```
credentials "app.terraform.io" {
  token = "your_tfc_token_here"
}
```

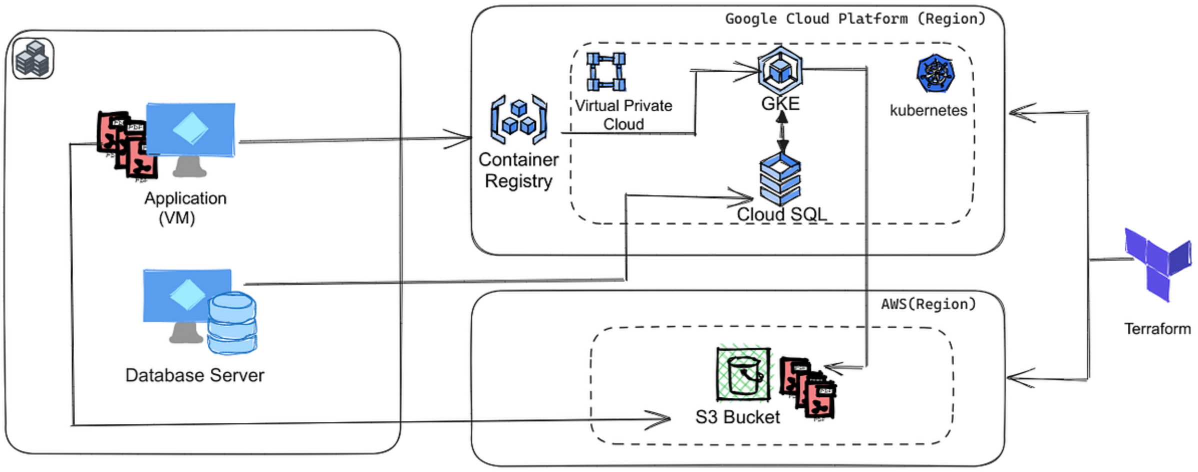
3.4 Private Module Registry

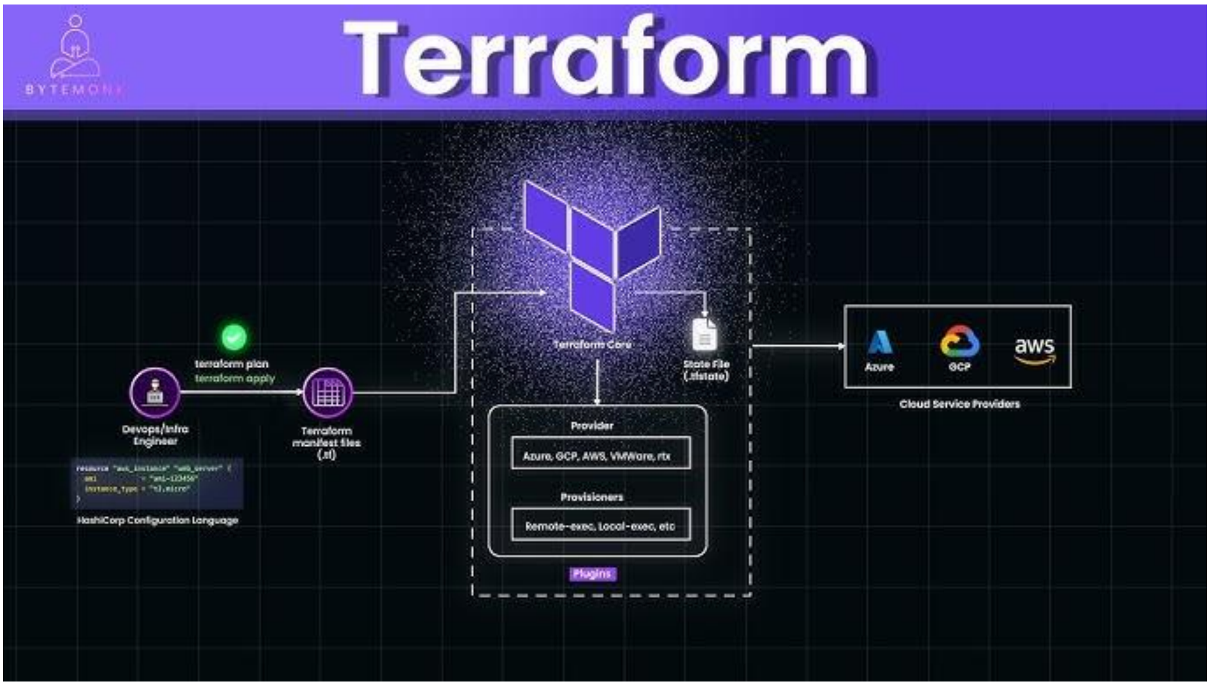
Allows teams to publish internal modules.

```
module "eks" {
  source  = "app.terraform.io/company/eks/aws"
  version = "1.2.0"
}
```

---

4. Deploying Multi-Cloud Environments





```
resource "aws_instance" "web_server" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Terraform is **cloud-agnostic**, supporting:

- AWS
- Azure
- GCP

- VMware
- Kubernetes
- OCI
- OpenStack
- Alibaba Cloud
- Multi-cloud federated systems

4.1 Example Multi-Cloud Project Structure

```
multi-cloud/
  aws/
    main.tf
  azure/
    main.tf
  gcp/
    main.tf
  modules/
    compute/
    networking/
```

4.2 Example: Create AWS EC2 + Azure VM

AWS: EC2 Instance

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

resource "aws_instance" "web" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Azure: VM

```
provider "azurerm" {
  features {}
}


resource "azurerm_resource_group" "rg" {
  name     = "prod-rg"
  location = "eastus"
}
```

4.3 Multi-Cloud Use Cases

Use Case	**Why Multi-Cloud?**
DR & BCP	Backup on another cloud
Hybrid workloads	Only some services needed from each cloud
Vendor lock-in prevention	Avoid dependency on single provider
Geo-redundancy	Better global performance

4.4 Multi-Cloud Best Practices

✓ Use modules to standardize across clouds
✓ Maintain separate **state files** per cloud
✓ Enforce policies using Open Policy Agent or Sentinel
✓ Use CI/CD pipelines for automation
✓ Ensure strong tagging strategy
✓ Use GitOps + Terraform Cloud for drift detection

Final Summary — Module 4

Topic	**Key Takeaways**
Modules & Workspaces	Reusable code + isolated environments
Remote State	Central storage with locking & collaboration
Terraform Cloud	VCS integration, remote runs, policies, registry
Multi-Cloud Deployment	Build AWS, Azure, GCP infra in one workflow

---