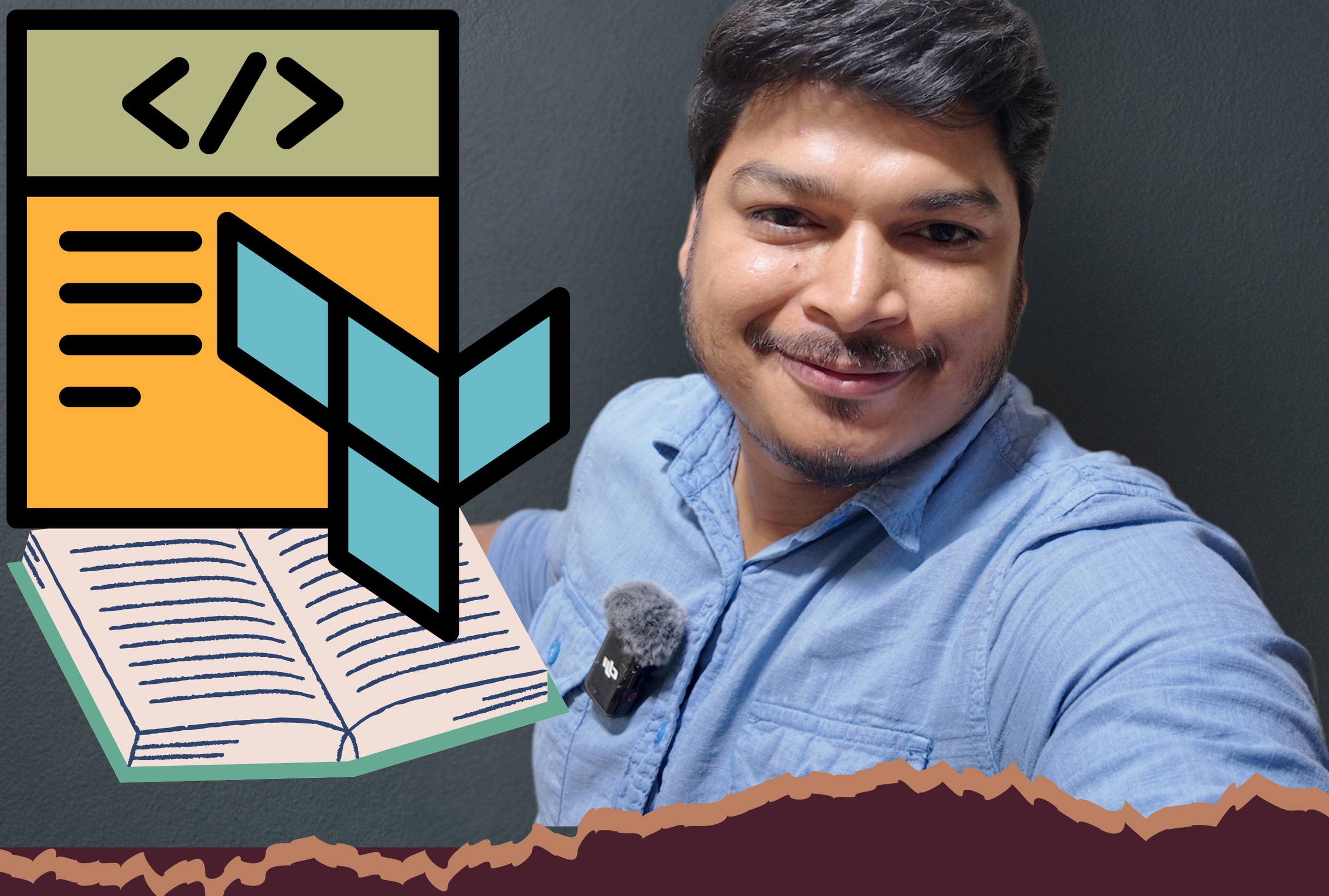


TERRAFORM MASTERBOOK

From Beginner to Expert

For Cloud & DevOps Engineers

By :

SANDIP DAS
LearnXOps.com

Table of Contents

ABOUT ME

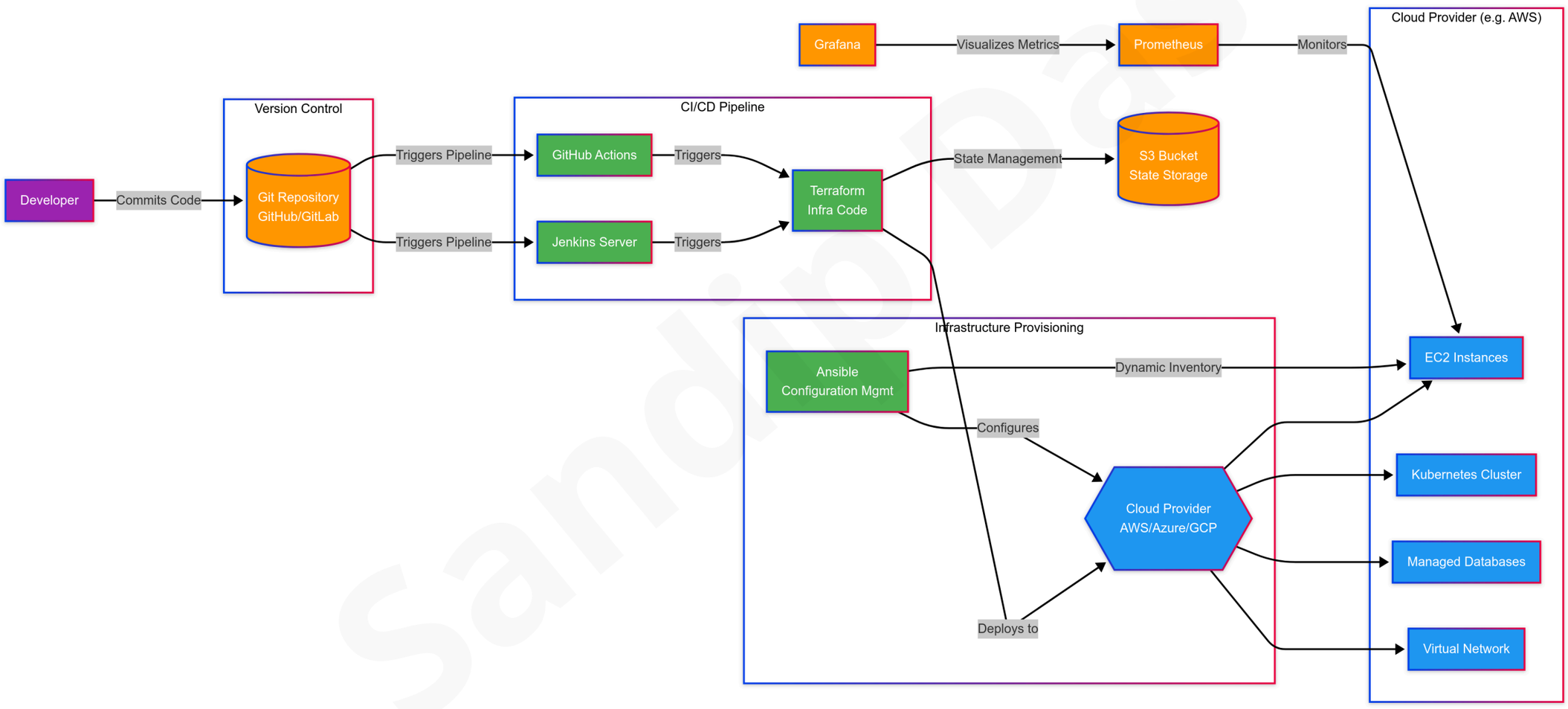
Hi! I’m Sandip Das, a Senior Cloud, DevOps & MLOps Engineer with 12+ years of experience, primarily working on production-grade Infrastructure as Code (IaC) using Terraform with AWS + Kubernetes (AWS EKS). Around 15–20% of my daily work revolves around Terraform — designing reusable modules, automating infrastructure provisioning, and optimizing cloud resource management for clients across the US, UK, EU, and UAE.

LearnXOps.com

Introduction

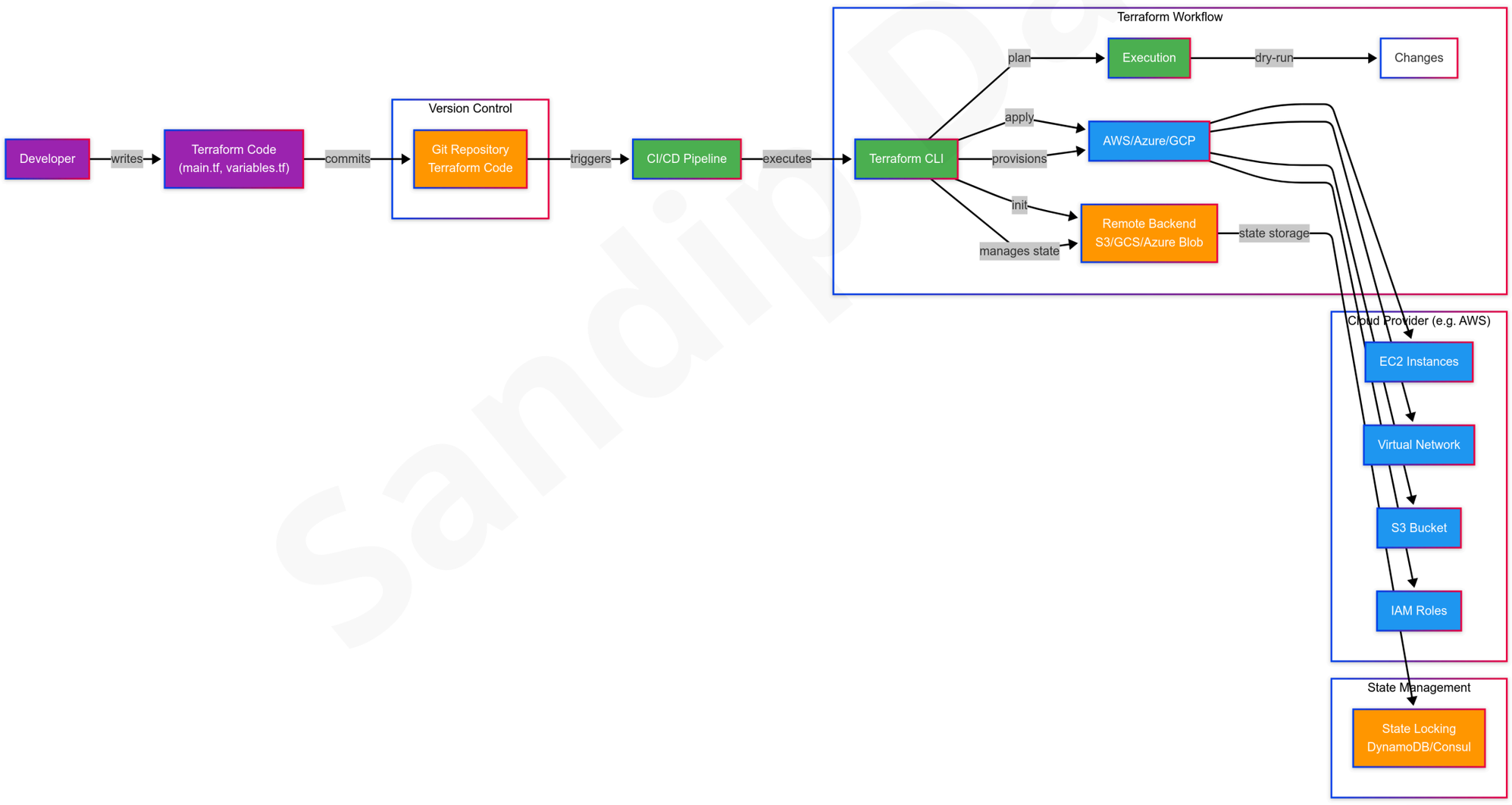
What is Infrastructure as Code ?

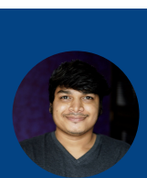
Infrastructure as Code (IaC) is the practice of provisioning and managing cloud infrastructure using code instead of manual processes. It brings automation, repeatability, and version control to infrastructure management, just like software development.



What is Terraform?

Terraform is an open-source Infrastructure as Code (IaC) tool by HashiCorp that lets you define and provision cloud infrastructure using a declarative language. It works across multiple cloud providers like AWS, Azure, and GCP, making your infrastructure predictable and version-controlled.





@Sandip Das

Introduction

Terraform vs. CloudFormation, ARM, Pulumi

Feature/Aspect	Terraform	CloudFormation	ARM Templates	Pulumi
Tool Type	Open-source IaC tool by HashiCorp	Native AWS IaC service	Native Azure IaC service	IaC tool using general-purpose languages
Language	HashiCorp Configuration Language (HCL)	JSON/YAML	JSON (Bicep is new, friendlier DSL)	TypeScript, Python, Go, .NET (C#, F#)
Multi-Cloud Support	✅ Yes	❌ AWS only	❌ Azure only	✅ Yes
Modularity & Reusability	✅ High (modules, workspaces)	Moderate (nested stacks, macros)	Limited (better with Bicep)	✅ High (code-native, packages, functions)
State Management	✅ External (Terraform Cloud, S3, etc.)	✅ Managed internally	✅ Managed internally	✅ Optional (uses backends like S3 or Pulumi Service)
Drift Detection	⚠ Manual (via terraform plan)	✅ Built-in	✅ Built-in	✅ With CLI tools and automation
Loops & Conditions	✅ Supported (limited via for_each, count, dynamic)	⚠ Limited	⚠ Verbose	✅ Native via programming logic
Learning Curve	Moderate (HCL is simple, declarative)	Easy for AWS users	Steep (JSON); Bicep is easier	Steep (depends on language, but powerful)
Community & Ecosystem	✅ Huge (providers, modules, plugins)	✅ Strong (AWS-native)	Moderate (Azure-focused)	Growing (especially for dev-heavy teams)
Use Case Fit	Best for multi-cloud, infrastructure-heavy, reusable IaC	Best for AWS-only, deep integrations	Best for Azure-only, native control	Best for developer-heavy teams, app+infra together

Installation and Setup

Linux (Debian/Ubuntu)

```
wget -O - https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform
```

macOS

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

Windows

```
choco install terraform
terraform -v
```

Or Download from here

Check Version After Installation

```
terraform -v
```

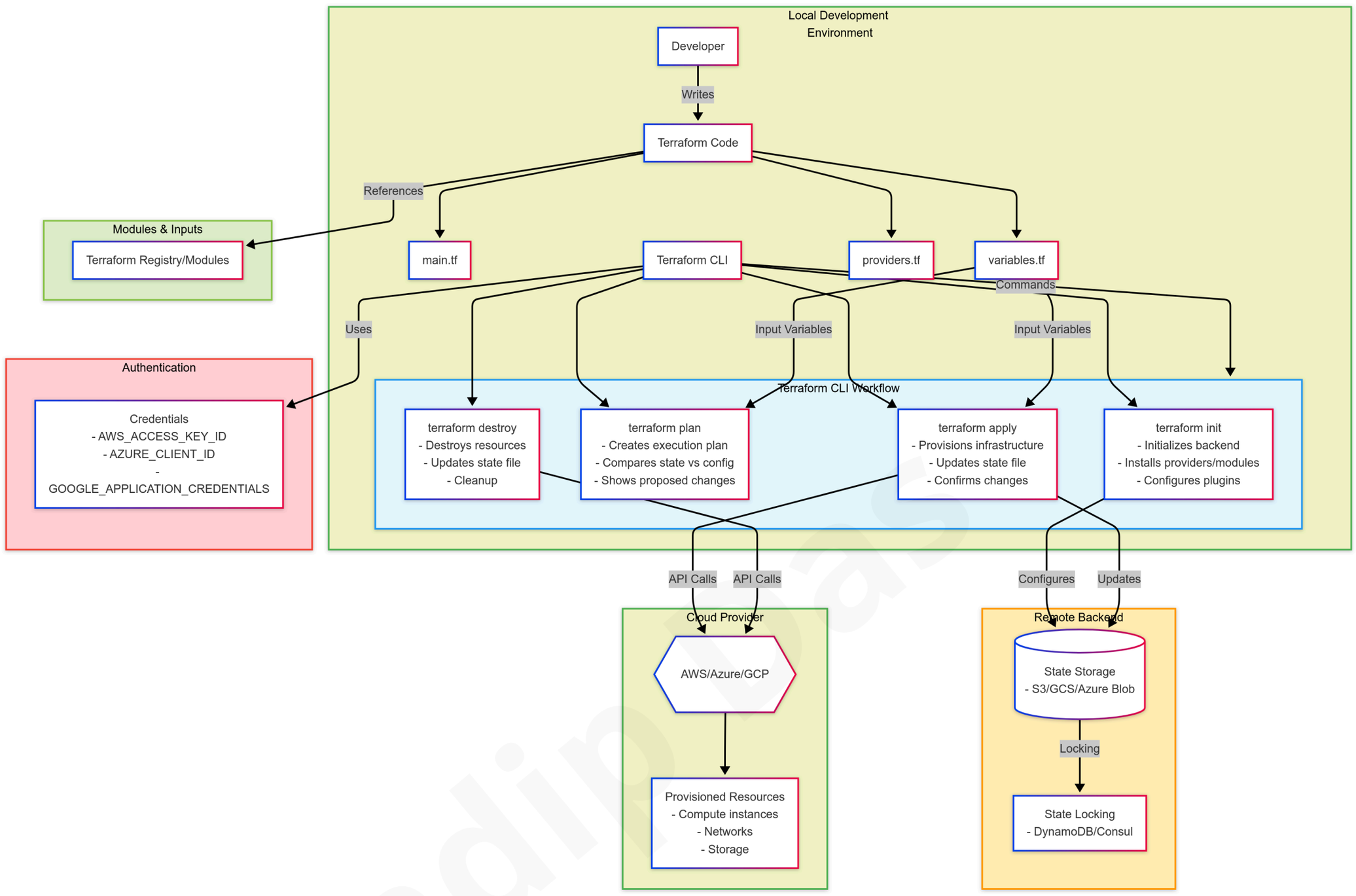
Terraform Version Managed (tfenv)

tfenv is a CLI tool that helps you easily manage and switch between multiple Terraform versions on your system.

Install tfenv (Linux/macOS)

```
git clone https://github.com/tfutils/tfenv.git ~/.tfenv
echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc

# or for Zsh:
# echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> ~/.zshrc
# source ~/.zshrc

# Install specific Terraform version
tfenv install 1.6.6

# Set global or local version
tfenv use 1.6.6
# or
# tfenv use 1.6.6 --global

# List installed and available versions
tfenv list
```


How Terraform CLI Works ?



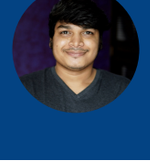
Terraform CLI reads your .tf files to understand desired infrastructure.

- **terraform init** sets up backend, providers, and modules.
- **terraform plan** shows changes Terraform will make by comparing current state vs config.
- **terraform apply** creates/updates resources via API calls and updates the state.
- **terraform destroy** removes infrastructure and updates the state file accordingly.

.tf files written in HCL

HCL stands for **HashiCorp Configuration Language** — it's what you use to write your Terraform code.

It's super readable, kind of like JSON but way easier on the eyes and built for defining infrastructure



@Sandip Das

Set-up Environment

AWS Setup

Install AWS CLI & Configure Credentials:

```
aws configure
# Provide AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, region
```

AZURE Setup

Install Azure CLI & Login:

```
az login
az account set --subscription "<SUBSCRIPTION_ID>"
```

GCP Setup

Install gcloud CLI & Authenticate

```
gcloud auth login
gcloud config set project <PROJECT_ID>
```


Core Concepts

Providers

Terraform providers are plugins that let Terraform talk to external platforms like AWS, Azure, GCP, GitHub, and more.
They expose resources and data sources so you can create and manage infrastructure through code.

e.g. , the AWS provider (hashicorp/**aws**) allows **managing AWS resources**, Azure's provider (hashicorp/**azurerm**) **manages Azure resources**, and so on. Providers must be declared and configured (e.g., setting the target region or credentials). A simple provider configuration looks like:

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

Popular Providers: aws, azurerm, google cloud, alibaba cloud, Oracle Cloud, kubernetes, helm, hashicorp/random, vault, cloudflare, datadog, newrelic, docker, gitlab, github, consul, mysql, postgresql, vsphere, openstack, alicloud, digitalocean

Get full list here

Resources

Resources in Terraform are a resource block describes an infrastructure object to create (VM, container cluster, database, etc.).

It simply means that the actual infrastructure components you want to create—like EC2 instances, S3 buckets, or VPC etc.
You define them in your .tf files, and Terraform takes care of provisioning them for you.
It has a type (e.g., aws_instance for an EC2 VM, azurerm_resource_group for an Azure resource group), a local name, and a set of parameters (arguments) defining its properties. e.g. an AWS EC2 instance resource might specify the AMI ID and instance type:

```
resource "aws_instance" "web" {
  ami           = "ami-0abcdef1234567890"
  instance_type = "t2.micro"
  tags = {
    Name = "HelloWorld"
  }
}
```

Core Concepts

Modules

Modules in Terraform are reusable chunks of code that group related resources together like a container for multiple resources that are used together

They help keep your configs clean, DRY (Don’t Repeat Yourself), and easy to manage across environments.

Fun fact, **every Terraform configuration is implicitly a module** (the “**root module**”), and you can create child modules to encapsulate complex pieces of infrastructure as reusable components.

For example, a “**VPC module**” could create network, subnets, and security groups; a “**compute module**” could create an auto-scaling group with load balancer. Modules encourage DRY (Don’t Repeat Yourself) code – you define it once and use it in many places. You can source modules from local paths, Git, or the Terraform Registry. For instance, to reuse a community AWS VPC module:

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.5.0"
  cidr    = "10.0.0.0/16"
  azs     = ["us-east-1a", "us-east-1b"]
  name    = "my-vpc"
}
```


Core Concepts

State

State in Terraform is a file that tracks the current setup of your infrastructure.
It helps Terraform know what's already been created so it can figure out what needs to change.

When you run terraform plan or apply, Terraform checks the real-time status of resources via cloud APIs and updates the state. Store state securely—ideally in remote backends—for team use and to protect sensitive data like resource IDs and secrets.

The state file also allows Terraform to detect drift and destroy or update the correct resources. Here's sample terraform.tfstate file and explanation of it:

```
{
 "version": 4,
 "terraform_version": "1.5.0",
 "resources": [
  {
   "mode": "managed",
   "type": "aws_instance",
   "name": "web",
   "provider":
"provider[\"registry.terraform.io/hashicorp/aws\"]",
   "instances": [
    {
     "schema_version": 1,
     "attributes": {
      "id": "i-0abcd1234efgh5678",
      "ami": "ami-1234567890abcdef0",
      "instance_type": "t2.micro",
      "tags": {
       "Name": "web-server"
      },
      "availability_zone": "us-east-1a",
      "public_ip": "54.123.45.67"
     }
    }
   ]
  }
 ]
}
```

version & terraform_version: Track the format and which Terraform version wrote this file.

resources: List of all managed infrastructure resources.

type: Type of resource (aws_instance).

name: Name used in your config (web).

id: Unique ID of the resource on the cloud (like EC2 instance ID).

attributes: All important resource details—like AMI, instance type, IP, tags, etc.

Terraform State Commands:

Display all resources tracked in the state file
terraform state list

Show detailed attributes of a resource in the state
terraform state show <resource_name>

Unlink a resource from Terraform without deleting it from the cloud
terraform state rm <resource_name>

Rename or moves a resource within the state file
terraform state mv <old_resource_name> <new_resource_name>

create "prod"
terraform workspace new prod

switch back to dev
$ terraform workspace select dev

Import existing resource into state
terraform import <resource_name> <resource_id>

@Sandip Das

Core Concepts

State Backend

A state backend in Terraform is where the .tfstate file is stored—locally or remotely.
It manages how state is loaded, saved, and optionally locked to prevent conflicts during team operations. There's 2 types of Back-end:

Local Backend

- Stores the state file on the local disk.
- Best for single-user or testing setups.

Remote Backends

Used for team collaboration, locking, and secure state management.

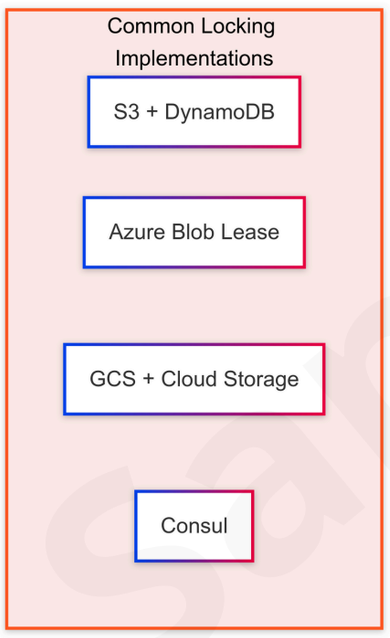
1. S3 (with DynamoDB for locking) – AWS
2. azurerm – Azure Blob Storage
3. gcs – Google Cloud Storage
4. consul – Uses Consul KV store with locking
5. http – Sends/receives state via HTTP endpoints
6. etcd – Useful in Kubernetes clusters
7. terraform cloud / enterprise – HashiCorp's hosted solution with built-in features

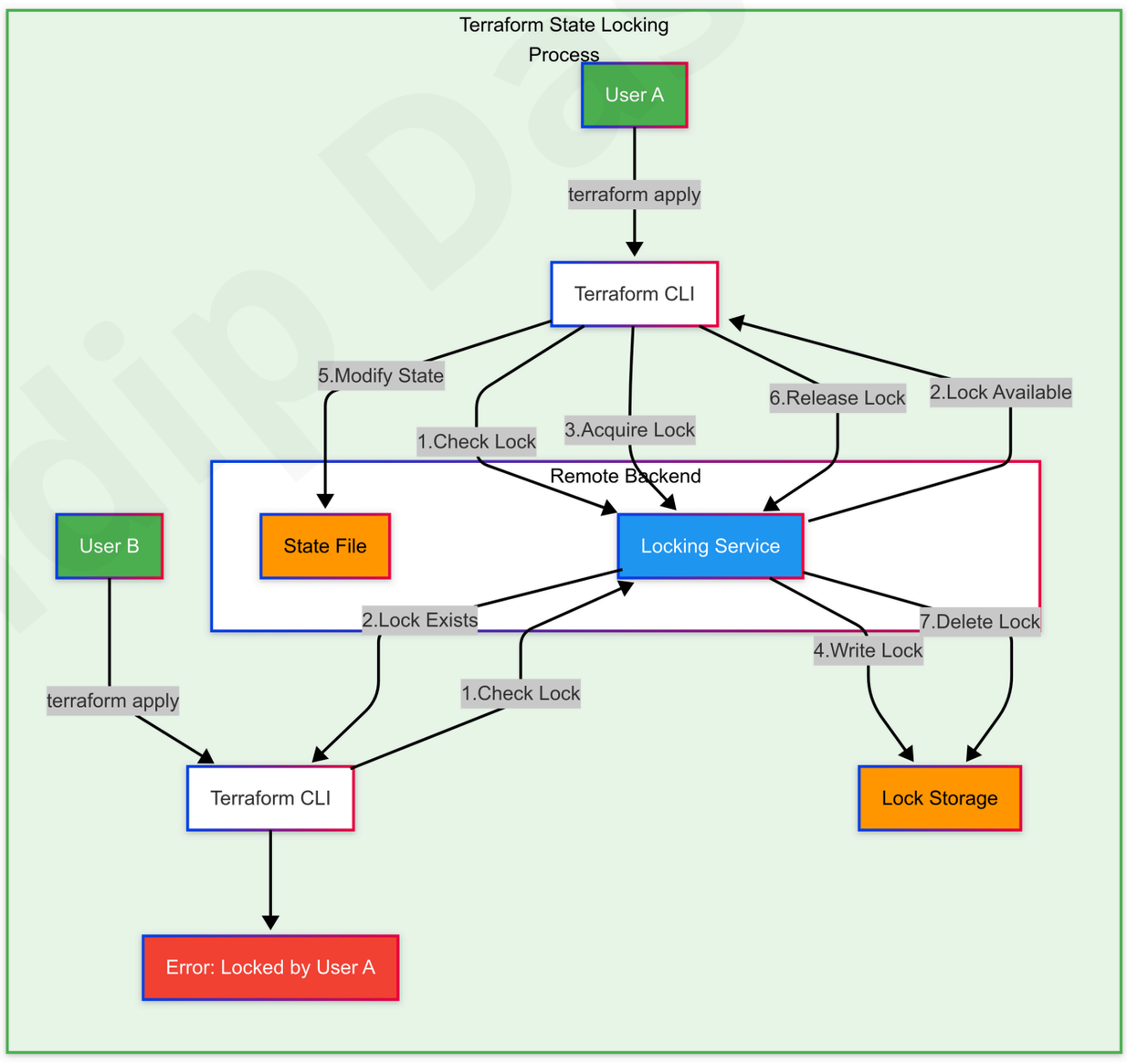
```
terraform {
  backend "s3" {
    bucket         = "my-terraform-state-bucket"
    key            = "env/dev/terraform.tfstate"
    region         = "us-east-1"
    dynamodb_table = "terraform-locks"
    encrypt        = true
  }
}
```

This Need to Preconfigured:
S3 Bucket (for storing the state file)
DynamoDB Table (for state locking)
Table name: terraform-locks
Primary key: LockID (string)

🔒 **State Locking & Consistency**

- State Locking prevents multiple users from running Terraform at the same time, avoiding conflicts.
- Remote backends like S3 + DynamoDB, or Terraform Cloud, support locking.
- Ensures consistent view of infrastructure, especially in team environments.
- Helps prevent race conditions, corrupted state, or unexpected infra drift.





@Sandip Das

Terraform Basics

Writing First Terraform Configuration

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

resource "aws_security_group" "web_sg" {
  name        = "web-sg"
  description = "Security group for web server"
  ingress = [{
    from_port   = 80,
    to_port     = 80,
    protocol    = "tcp",
    cidr_blocks = ["0.0.0.0/0"]  # open HTTP to the world
  }]
}

resource "aws_instance" "web" {
  ami             "ami-00400f5621e3abbf4"   # Amazon Linux 2 in us-east-1
  instance_type   "t2.micro"
  vpc_security_group_ids = [aws_security_group.web_sg.id]  # attach SG

  tags = {
    Name = "TerraformWebServer"
  }
}
```

This config does the following:

- **Defines the AWS provider** in us-east-1.
- **Creates a Security Group** (web_sg) that allows inbound TCP port 80 from anywhere (0.0.0.0/0). All other ports are implicitly blocked.
- **Launches an EC2 Instance (web)** using the specified AMI and instance type, and associates it with the security group we created.

Apply the Configuration:

- Run **terraform init** ,this will ensure the AWS provider is ready.
- Run **terraform plan** and review the plan. It should show 2 to add (one security group, one instance)
- Run **terraform apply** and type "**yes**" to confirm. Terraform will create the security group then the EC2 instance. Within seconds, you'll have a running VM on AWS.

You can also add an output to get the public IP, for example:

```
output "instance_ip" {
  value = aws_instance.web.public_ip
}
```

- **To verify**, bo to the AWS EC2 console, you should see the new instance running. The security group "web-sg" should be attached to it, with a rule allowing port 80. If you install a web server on the instance, it would be reachable on its public IP (for now, the instance has no key pair or user data – we're focusing on provisioning basics).

- Clean up by running **terraform destroy** when finished. Confirm and Terraform will terminate the EC2 and delete the security group, leaving no trace (and no ongoing cost)

@Sandip Das

Terraform Basics

📥 Input Variables

Used to parameterize Terraform configs for flexibility.

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

Types

- **string**
- **number**
- **bool**
- **list**(string)
- **map**(string)
- **object**({...})
- **any** (dynamic type)

⚠ Validation

```
variable "port" {
  type = number

  validation {
    condition     = var.port >= 1024 && var.port <= 65535
    error_message = "Port must be between 1024 and 65535."
  }
}
```

Default Values

- If not provided by the user, Terraform uses the default value.

This Terraform variable port accepts only numeric values between 1024 and 65535. The validation block ensures the port is within this range, else it throws an error with the given message.

Terraform Variable Loading Order (Precedence)

1. Default values in the variable block
2. Environment variables (e.g., TF_VAR_name)
3. terraform.tfvars or terraform.tfvars.json
4. Any *.auto.tfvars or *.auto.tfvars.json files
5. Explicit -var or -var-file options on CLI

📤 Output Variables

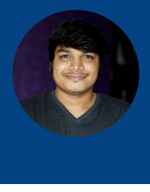
It is used to expose useful info after a Terraform apply.

```
output "instance_ip" {
  value       = aws_instance.web.public_ip
  description = "Public IP of the EC2 instance"
}
```

The command :
terraform output instance_ip
It prints the value of the instance_ip output variable defined in your Terraform config.

e.g.
$ terraform output instance_ip
54.210.123.45

```
module "web" {
  source = "./web"
}

output "web_ip" {
  value = module.web.instance_ip
}
```



@Sandip Das

Terraform Basics

Data Sources

What is a Data Source?

Data sources let Terraform read existing infrastructure or external data without creating it.
They're used when you want to reference things, not provision them.

How to Use a Data Block?

```
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*-x86_64-gp2"]
  }
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t2.micro"
}
```

Example:
Fetch the latest Amazon Linux 2 AMI in AWS

Common Use Cases

- 🔍 Get the latest AMI ID
- 🔐 Fetch secrets from Secret Manager
- 👨‍💻 Lookup current user/account/region
- 📦 Read existing VPCs, subnets, security groups
- 📊 Fetch remote JSON or Terraform outputs from other workspaces

Advanced Terraform

Custom Module Development

What Are Modules, again ?
A module is a container for multiple resources that are used together. A custom module is just a directory with Terraform files (.tf) that defines specific infrastructure logic, which you can reuse in other Terraform configs.

Writing and Using Custom Modules

```
terraform/
 ├── main.tf        # Root
module
 ├── variables.tf
 ├── outputs.tf
 ├── modules/
 │    └── s3_bucket/
 │        ├── main.tf
 │        ├── variables.tf
 │        └── outputs.tf
```

Using the Module in Root main.tf:

```
module "my_bucket" {
  source      = "./modules/s3_bucket"
  bucket_name = "my-custom-terraform-bucket"
  acl         = "public-read"
}

outputs.tf:
output "my_bucket_id" {
  value = module.my_bucket.bucket_id
}
```

Here's a Detailed Video on It

Writing a Custom Module (modules/s3_bucket)

modules/s3_bucket/variables.tf

```
variable "bucket_name" {
  type = string
}

variable "acl" {
  type    = string
  default = "private"
}
```

modules/s3_bucket/main.tf

```
resource "aws_s3_bucket" "this" {
  bucket = var.bucket_name
  acl    = var.acl
}
```

modules/s3_bucket/outputs.tf

```
output "bucket_id" {
  value = aws_s3_bucket.this.id
}
```

Best Practices

- Keep modules small and focused.
- Use source = "git::https://..." for remote modules.
- Version your modules if used across teams.
- Validate inputs with validation blocks.

Advanced Terraform

Terraform Workspaces

What Are Terraform Workspaces?

Terraform workspaces allow you to manage multiple environments (e.g., dev, staging, prod) using the same configuration—but with separate state files.

Terraform Worksaces commands:

List all workspaces
terraform workspace list

Create a new workspace
terraform workspace new dev

Switch to an existing workspace
terraform workspace select dev

Show the current workspace
terraform workspace show

💡 **How They Work**
Each workspace has its own state file stored under .terraform/:

terraform.tfstate.d/<workspace_name>/terraform.tfstate

Using Workspaces in Code:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-bucket-${terraform.workspace}"
  acl    = "private"
}
```

Output when using dev workspace:
my-bucket-dev

🚫 **Workspace Limitations:**

- Not a replacement for full multi-env strategies.
- Doesn't isolate backends or providers—still share config unless explicitly parameterized.
- Use with care in production, especially with remote backends.

✅ **When to Use**

- Simple environment isolation (dev/stage/prod) without duplicating code.
- Lightweight experimentation without affecting the main state.

Advanced Terraform

Terraform Core Functions & Expressions

for expression

Used to iterate over a list or map and transform it.

```
variable "names" {
  default = ["dev", "stage", "prod"]
}

output "env_names" {
  value = [for name in var.names : upper(name)]
}
```

Output: ["DEV", "STAGE", "PROD"]

if expression (Conditional)

Conditional logic using ? : similar to ternary operators.

```
variable "env" {
  default = "prod"
}

output "instance_type" {
  value = var.env == "prod" ? "t3.large" :
"t3.micro"
}
```

lookup function

Safely get a value from a map.
hcl

```
variable "ami_map" {
  default = {
    "us-east-1" = "ami-1234"
    "us-west-2" = "ami-5678"
  }
}

output "ami_id" {
  value = lookup(var.ami_map, "us-west-2", "ami-
default")
}
```

merge function

Combine multiple maps.

```
output "combined_tags" {
  value = merge(
    { Name = "AppServer" },
    { Environment = "prod" },
    { Owner = "team-devops" }
  )
}
```

Example output:

```
{
  Name        = "AppServer"
  Environment = "prod"
  Owner       = "team-devops"
}
```

Complex Dynamic Blocks Example

e.g. Dynamically creates multiple ingress blocks.

```
variable "ingress_rules" {
  default = [
    { port = 22,  cidr = "0.0.0.0/0" },
    { port = 80,  cidr = "0.0.0.0/0" },
    { port = 443, cidr = "10.0.0.0/16" }
  ]
}

resource "aws_security_group" "web_sg" {
  name        = "web-sg"
  description = "Web Security Group"
  vpc_id      = "vpc-xxxxxxx"

  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = "tcp"
      cidr_blocks = [ingress.value.cidr]
    }
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Best Practices:

- Use locals to simplify and reuse complex expressions.
- Use dynamic blocks for repeatable nested structures, but avoid over-nesting.
- Use safe functions like lookup, try, coalesce to prevent errors.
- Test with terraform console and add comments for clarity.

@Sandip Das

Advanced Terraform

Provisioners

Provisioners are used to execute scripts or commands on a local or remote machine as part of a Terraform resource lifecycle (e.g., after creation).

Types of Provisioners:

file: Transfers files from your machine to the provisioned resource

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

local-exec : Runs a command locally (on the machine running Terraform).

```
 provisioner "local-exec" {
  command = "echo Deployment complete > output.log"
}
```

remote-exec: Runs a command on the remote resource (e.g., EC2) using SSH/WinRM.

```
provisioner "remote-exec" {
  inline = ["sudo apt update", "sudo apt install nginx -y"]
}
```


Important Notes

- Run during create or destroy lifecycle stages.
- Can fail deployments if the script or connection fails.
- Not idempotent — changes aren't tracked by Terraform state.

Alternatives to Provisioners (Recommended)

✅ User data (for AWS EC2) to bootstrap instances.
✅ Cloud-init for instance initialization.
✅ Configuration Management tools (Ansible, Chef, Puppet).
✅ Startup scripts in resource metadata (like GCP metadata_startup_script).

Advanced Terraform

Terraform Meta-Arguments

Meta-arguments are special keywords in Terraform that modify how resources behave, but are not part of the resource itself. They're used to control resource creation, repetition, and lifecycle.

Common Meta-Arguments:

depends_on: Explicitly declare resource dependencies.

```
resource "aws_instance" "app" {
  ami           = "ami-1234"
  instance_type = "t2.micro"
  depends_on    = [aws_security_group.sg]
}
```

count: Used to create multiple copies of a resource (using index).

```
resource "aws_instance" "web" {
  count         = 3
  ami           = "ami-1234"
  instance_type = "t2.micro"
}
```

for_each: Create multiple resources from a map or set of strings (with named keys).

```
resource "aws_s3_bucket" "buckets" {
  for_each = toset(["logs", "images", "backups"])
  bucket   = "my-${each.key}-bucket"
}
```

lifecycle : Control how Terraform handles create, update, delete operations.

```
resource "aws_instance" "app" {
  ...
  lifecycle {
    prevent_destroy = true
    ignore_changes  = [tags["Name"]]
    create_before_destroy = true
  }
}
```

- **prevent_destroy**: blocks accidental deletion
- **ignore_changes**: skip re-apply on certain changes
- **create_before_destroy**: avoids downtime during replacement

Error Handling & Debugging

📋 Terraform Logs

Enable Debug Logging:

Use the **TF_LOG** environment variable to control log level:

export TF_LOG=DEBUG
terraform apply

Levels: TRACE > DEBUG > INFO > WARN > ERROR

To write logs to a file:
export TF_LOG_PATH="terraform.log"

Common Error Patterns:

Dependency Cycles:
Error: Cycle: aws_instance.web, aws_security_group.sg
Reason: Resource A depends on B and B depends back on A.
Fix: Use depends_on to break or manage the dependency clearly.

Incorrect for_each or count:
Error: Invalid for_each argument
Reason: Using null, map with duplicate keys, or a non-iterable.
Fix: Ensure for_each is a set, list, or map and properly formatted.

Provider/Plugin Errors:
Error: Failed to instantiate provider
Reason: Missing or misconfigured provider block.
Fix: Run terraform init and check provider version/config.

Permission Denied (403/AccessDenied):
Reason: Terraform can’t create/update/delete due to IAM or API restrictions.
Fix: Check credentials, roles, and permissions in cloud provider.

State Locking Issues:
Error: Error acquiring the state lock
Reason: Another operation is using the state file.
Fix: Use terraform force-unlock <lock-id> only if safe.

Testing and CI/CD with Terraform

Terraform Testing

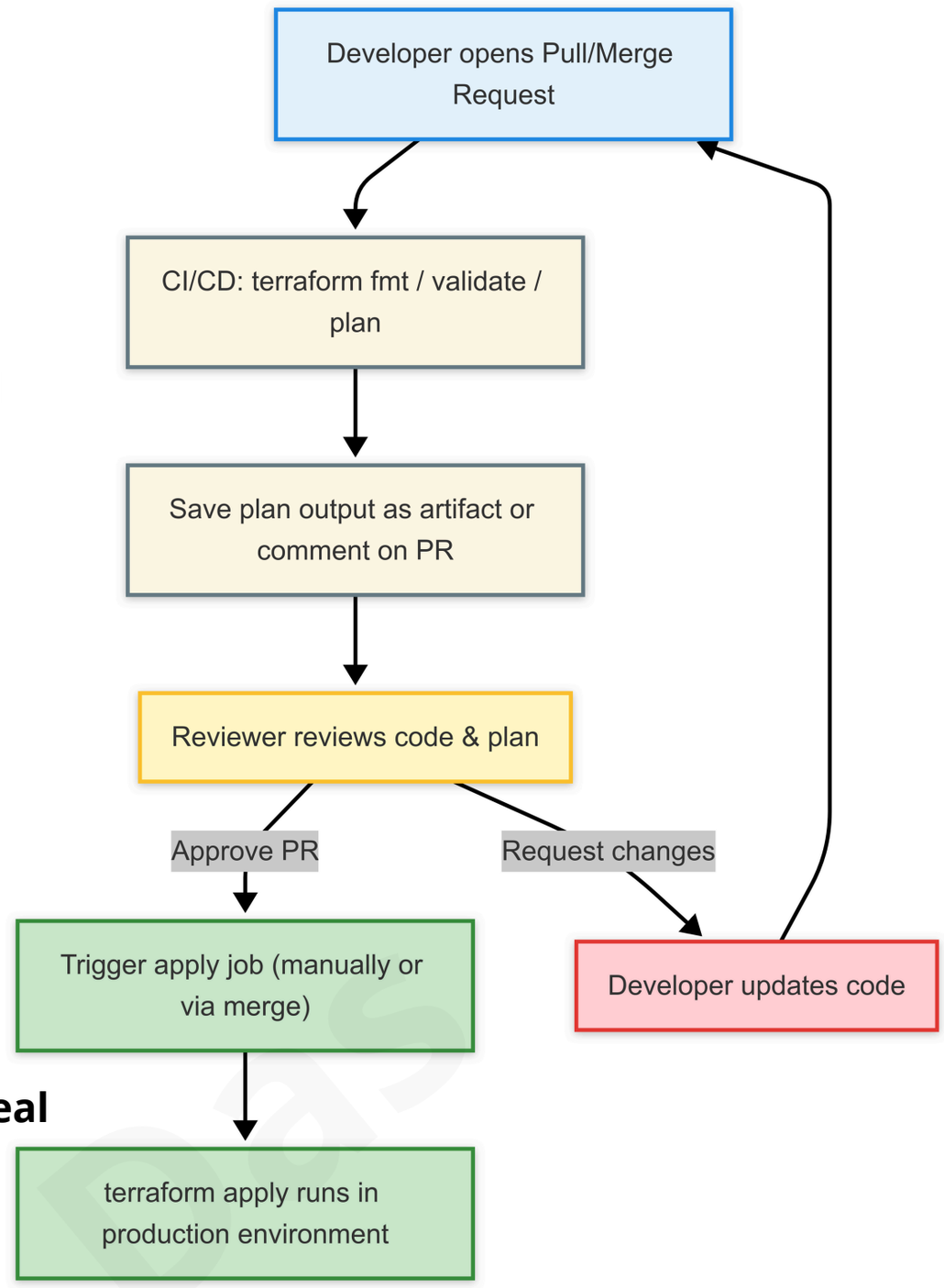
Unit Testing

terraform validate : It Checks syntax and internal logic.
tflint : Lints Terraform code for best practices and errors.

Policy & Security Testing:
Checkov: Scans Terraform code for security misconfigurations.
e.g. checkov -d .

InSpec : Test actual infrastructure state using compliance tests.Great for post-deployment auditing.

Terratest: Automated infrastructure tests using real Terraform deployments. Can assert resource outputs, availability, and behavior.



Terraform CI/CD with GitHub Actions:

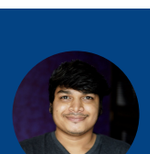
Use GitHub Environments to require manual approvals before running terraform apply

```yaml
jobs:
  terraform:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://console.aws.amazon.com/
    steps:
      - uses: actions/checkout@v2
      - uses: hashicorp/setup-terraform@v2
      - run: terraform init
      - run: terraform validate
      - run: terraform plan -out=tfplan
      - name: Terraform Apply
        run: terraform apply tfplan
        if: github.ref == 'refs/heads/main'
```

Terraform CI/CD with GitLab CI

Use when: manual for manual apply

```yaml
stages:
  - validate
  - plan
  - apply

terraform-validate:
  stage: validate
  script:
    - terraform init
    - terraform validate

terraform-plan:
  stage: plan
  script:
    - terraform plan -out=tfplan
  artifacts:
    paths:
      - tfplan

terraform-apply:
  stage: apply
  script:
    - terraform apply tfplan
  when: manual
  only:
    - main
```

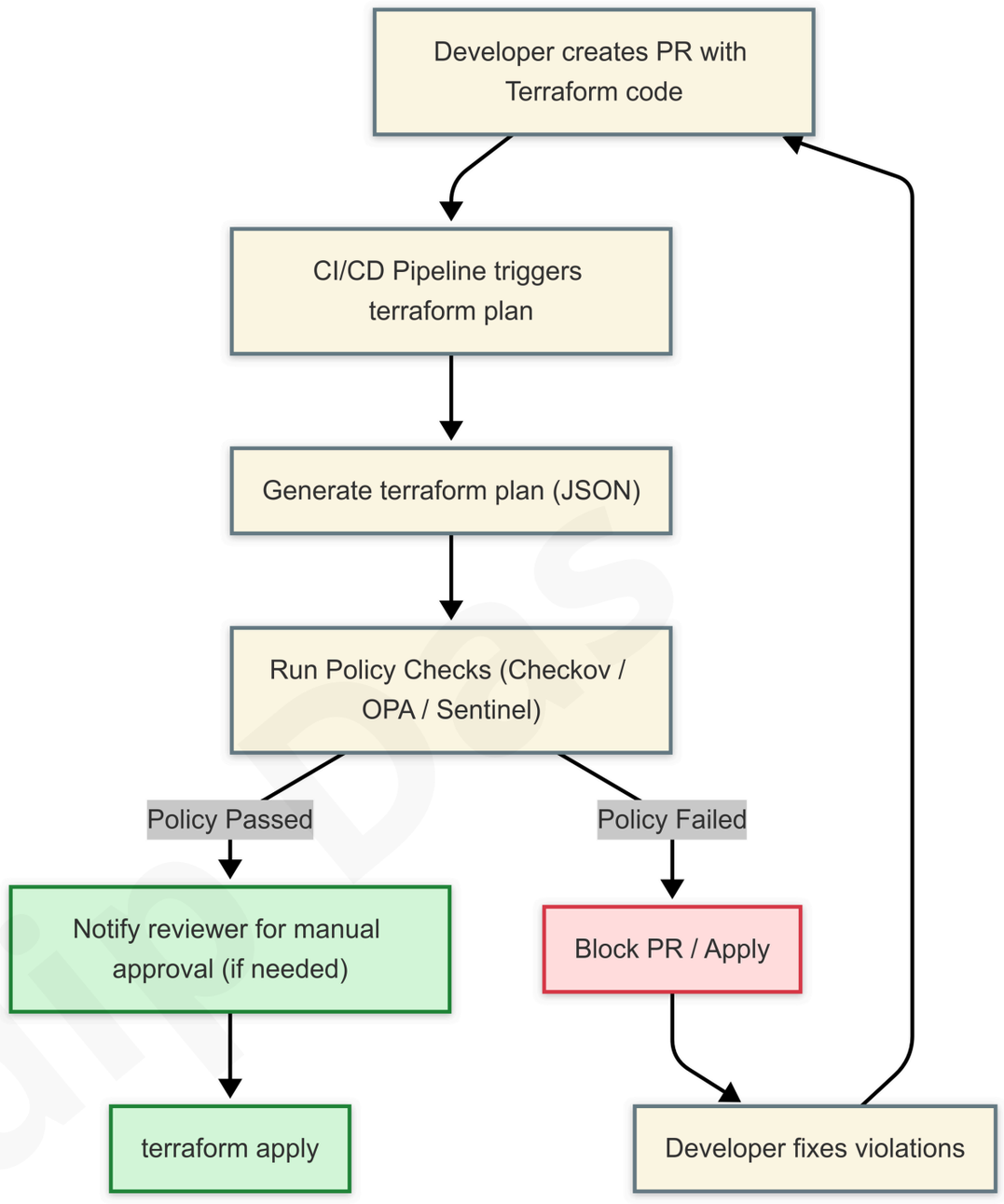



@Sandip Das

Policy as Code

Policy as Code in Terraform

Policy as Code means defining security, compliance, and governance rules in code — and enforcing them automatically during Terraform workflows.



Why Use Policy as Code?

🔒 Enforce security standards (e.g., no public S3 buckets)
📦 Standardize resource configurations across teams
🚫 Prevent misconfigurations before deployment
📄 Create audit trails for compliance

Common Tools for Policy as Code:

Sentinel (HashiCorp):

- Native to Terraform Cloud & Enterprise
- Policy language: custom DSL
- Enforces policies at plan, apply, and destroy stages

 # Sentinel example: deny public S3

```
import "tfplan"
main = rule {
  all tfplan.resources.aws_s3_bucket as _, r {
    r.applied.public_access_block.block_public_acls is true
  }
}
```

OPA + Conftest

- Open Policy Agent with Rego language
- Use with any CI/CD pipeline or locally
- Tool: conftest
- Run with: conftest test terraform.tfplan.json

```
# deny-public-s3.rego
package terraform.aws

deny[msg] {
  input.resource_type == "aws_s3_bucket"
  input.values.acl == "public-read"
  msg = "S3 bucket should not be publicly readable."
}
```

Policy Approval Flow:

terraform plan → exported as JSON
Policy engine (Sentinel/OPA/Checkov) runs checks
❌ If violation → block merge or apply
✅ If pass → proceed to apply or PR approval

Where to Integrate?

- Pre-commit hooks (pre-commit-terraform)
- CI/CD pipelines (GitHub Actions, GitLab CI)
- Terraform Cloud (with Sentinel)
- Atlantis or terragrunt workflows

@Sandip Das

Secrets Management

Terraform should never store or expose secrets in plain text. Here's how to handle secrets properly.

Different ways to handle secrets:

1) Environment Variables:
Use environment variables to store secrets like access keys, tokens, and sensitive variables.

```
export TF_VAR_db_password="supersecret"
```

In variables.tf:

```
variable "db_password" {
  type      = string
  sensitive = true
}
```

Keeps secrets out of code and Terraform state files are masked if sensitive = true.

2) Vault Integration:
We can use HashiCorp Vault to centrally manage and pull secrets dynamically.

```
provider "vault" {
  address = "https://vault.example.com"
}

data "vault_kv_secret_v2" "db_creds" {
  mount = "secret"
  name  = "db/creds"
}

output "db_user" {
  value     = data.vault_kv_secret_v2.db_creds.data["username"]
  sensitive = true
}
```

Best Practices:

- Use .gitignore to ignore .tfstate and .tfvars files.
- Use terraform.tfvars for local testing only — never commit secrets.
- Use **sops + age** or **gpg** to **encrypt .tfvars** files if needed.
- Prefer remote state with encryption at rest (e.g., S3 + KMS).
- Use Terraform Cloud/Enterprise or CI Secrets Manager (e.g., GitHub/GitLab secrets).

@Sandip Das

Multi Cloud

We have already cover one AWS Project at the start, now lets , explore other clouds

Azure Project – Provision a Linux VM

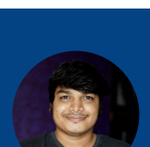
Scenario: Deploy a Linux virtual machine in Azure. Unlike AWS, Azure requires setting up a Resource Group and virtual network for the VM. Terraform will help create all necessary components: a resource group, virtual network, subnet, network interface, and the VM itself.

Prerequisites: An Azure account with CLI or service principal credentials configured. Azure authentication for Terraform is typically done via environment variables (ARM_CLIENT_ID, ARM_CLIENT_SECRET, etc. for a service principal) or Azure CLI login. Make sure you have these ready. Decide on an Azure region (e.g., eastus).

```hcl
provider "azurerm" {
  features {}  # enables AzureRM provider with default features
}

resource "azurerm_resource_group" "rg" {
  name     = "terraform-rg"
  location = "East US"
}

resource "azurerm_virtual_network" "vnet" {
  name                = "terraform-vnet"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  address_space       = ["10.0.0.0/16"]
}

resource "azurerm_subnet" "subnet" {
  name                 = "default"
  resource_group_name  = azurerm_resource_group.rg.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes     = ["10.0.1.0/24"]
}

resource "azurerm_network_interface" "nic" {
  name                = "vm-nic"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location

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

resource "azurerm_linux_virtual_machine" "vm" {
  name                = "terraform-vm"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  size                = "Standard_B1s"              # small VM size
  admin_username      = "azureuser"
  admin_password      = "P@ssword1234!"             # example password (Weak; use SSH keys or
stronger password in real setups)
  network_interface_ids = [azurerm_network_interface.nic.id]

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "18.04-LTS"
    version   = "latest"
  }
}
```

Let's break down what happens here:

- We use the AzureRM provider (with features {} block – this is required to use the provider, even if no features configured).
- Resource Group: A container in Azure for resources. We create one called "terraform-rg" in East US region.
- Virtual Network and Subnet: Azure resources need a VNet. We create a VNet with address space 10.0.0.0/16, and within it a Subnet 10.0.1.0/24. This is where the VM's network interface will reside.
- Network Interface: We create a NIC named "vm-nic" in the subnet. We're not adding a public IP in this simple setup (so the VM will be internal-only by default). Also not adding Network Security Group rules – Azure, by default, allows all outbound and denies inbound unless opened. For simplicity, we omit NSG (security rules).
- Linux Virtual Machine: We create a VM with Ubuntu 18.04 image. We specify an admin username and password (note: using passwords is not recommended; SSH keys or Azure Key Vault would be better for real environments). The VM is attached to the NIC and thus to the VNet. The size is the VM SKU (Standard_B1s is a small VM type).



@Sandip Das

Multi Cloud

GCP Project – Provision a GKE Kubernetes Cluster

Scenario: On Google Cloud Platform, we'll create a Google Kubernetes Engine (GKE) cluster using Terraform. This will demonstrate Terraform managing a managed service (Kubernetes) on GCP.

Prerequisites: A GCP project with the GKE API enabled, and credentials set up (for example, a service account JSON key and GOOGLE_APPLICATION_CREDENTIALS environment variable, or use gcloud auth application-default login). Know the project ID and choose a region or zone for the cluster.

```
provider "google" {
  project = "<YOUR_GCP_PROJECT_ID>"
  region  = "us-central1"
}

resource "google_container_cluster" "primary" {
  name     = "terraform-cluster"
  location = "us-central1-a"   # a specific zone for the cluster
  initial_node_count = 2

  # (Using basic auth - deprecated on GKE; in real setups, use OAuth or remove this)
  master_auth {
    username = "admin"
    password = "admin-password"
  }

  node_config {
    machine_type = "e2-medium"
  }
}
```

Let's break down what happens here:

- The Google provider is configured with your project ID and region (us-central1). We specify a zone in the cluster resource (us-central1-a) because GKE can be regional or zonal – here we choose a single zone cluster for simplicity.
- google_container_cluster defines a GKE cluster named "terraform-cluster" with 2 nodes. We include a node_config to specify the machine type for nodes. We also set a basic username/password for the Kubernetes master (GKE is deprecating this in favor of client certificates/OAuth; but for a tutorial, it's okay).
- This single resource will create the cluster and the default node pool.

Multiple Cloud Together:

Scenario: Same project using AWS & Azure

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

provider "azurerm" {
  alias    = "azure_east"
  features = {}
}

resource "aws_s3_bucket" "aws_bucket" {
  provider = aws.aws_us
  bucket   = "multi-cloud-aws-bucket"
}

resource "azurerm_resource_group" "azure_rg" {
  provider = azurerm.azure_east
  name     = "multi-cloud-rg"
  location = "East US"
}
```

Multi Cloud Best practices

- **Use provider aliases**
- **Organize code with modules per cloud**
- Use terraform workspace or backend per cloud for state separation
- Use remote state backends (e.g., S3, GCS, Azure Blob)
- Isolate cloud-specific logic into modules
- Use common tags/labels to unify resource visibility
- Manage shared infra (DNS, CDN, auth) centrally

@Sandip Das

GET SET GO

Keep Learning, Keep Building! 🚀

Congratulations on completing Terraform Masterbook - From Beginner to Expert!

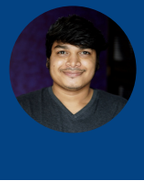
This is just the beginning of your journey in the world of Infrastructure as Code and cloud automation.

The tech landscape is ever-evolving, and the best way to stay ahead is to keep learning, experimenting, and sharing knowledge with others. Whether you are provisioning your first cloud resource or managing complex, multi-cloud infrastructure, remember: every expert was once a beginner.

💡 **What's next?**

- Apply what you've learned by working on real-world infrastructure projects.
- Contribute to open-source Terraform modules and collaborate with the DevOps community.
- Stay updated with the latest in Terraform, cloud providers, and IaC best practices through blogs, conferences, and forums.
- Teach and mentor others – knowledge grows when shared!

If this book helped you, let me know! Connect with me on LinkedIn or follow my work at LearnXOps.
Happy learning, and may your plans always apply successfully! 🚀🔥
- Sandip Das



@Sandip Das