Types of IaC Tools

AVIZ ACADEMY

docker

ANSIBLE

HashiCorp Terraform

SALTSTACK

puppet

CloudFormation

OpenTofu

Pulumi

Types of IaC Tools

AVIZ ACADEMY

Configuration Management

ANSIBLE

puppet

SALTSTACK

Server Templating

docker

HashiCorp Vagrant

Provisioning Tools

HashiCorp Terraform

CloudFormation

Pulumi

Types of IaC Tools

AVIZ ACADEMY

Configuration Management

ANSIBLE

puppet

SALTSTACK

- Designed to Install and Manage Software
- Maintains Standard Structure
- Version Control
- Idempotent

Types of IaC Tools

Server Templating Tools

- Pre Installed Software and Dependencies
- Virtual Machine or Docker Images
- Immutable Infrastructure

Types of IaC Tools

Provisioning Tools

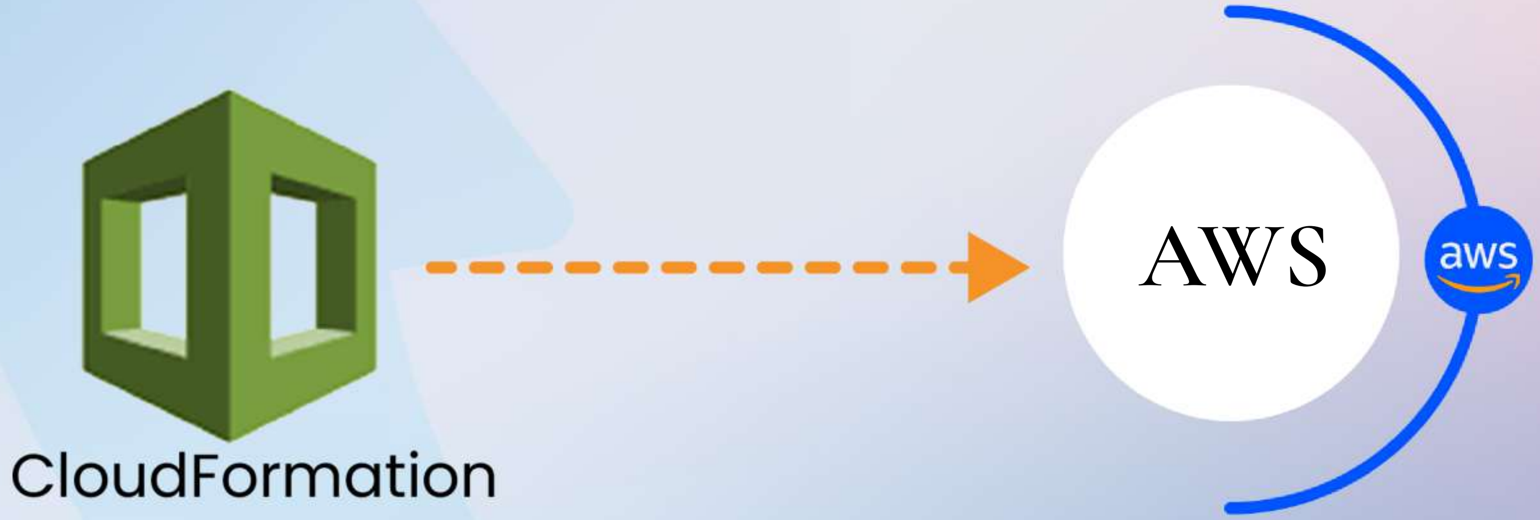
HashiCorp Terraform

CloudFormation

Pulumi

- Deploy Immutable Infrastructure resources
- Servers, Databases, Network Components etc.
- Multiple Providers

Which IaC Tools Should I Use?



Which IaC Tools Should I Use?

AVIZ ACADEMY

HashiCorp
Terraform

Physical Machines

VMware

AWS

GCP

Azure

What is Terraform?

Terraform is a powerful infrastructure automation tool that revolutionised how organisations build and manage their IT infrastructure. Created by Mitchell Hashimoto in 2014, it has become the industry standard for Infrastructure as Code (IaC), enabling teams to define, provision, and manage infrastructure through declarative configuration files rather than manual processes

Terraform: Key Characteristics and Capabilities

Infrastructure as Code

Terraform allows you to define Infrastructure using HashiCorp Configuration Language (HCL), written in Go. This declarative approach enables version control, collaboration, and repeatable deployments across environments.

Cloud Agnostic Platform

Platform-independent and cloud agnostic, Terraform works seamlessly with AWS, Azure, Google Cloud, and hundreds of other providers. It can even manage cross-cloud dependencies and on-premises infrastructure simultaneously.

Ownership and Licensing

Originally developed by HashiCorp and now maintained by IBM, Terraform transitioned from open source to the Business Source Licence after version 1.6, reflecting its maturity and enterprise adoption.

This combination of flexibility, power, and cross-platform capability makes Terraform the go-to choice for IT professionals seeking to automate and standardise infrastructure provisioning at scale.

Terraform transforms infrastructure code into real cloud resources through a systematic orchestration process. By reading your configuration files, planning changes, and managing state, Terraform Core acts as the intelligent bridge between your infrastructure-as-code and your production environment.

Infrastructure Code

- **Template:** Terraform configuration files $(*.\text{tf})$ written in HashiCorp Configuration Language
- **Scripts:** Infrastructure-as-code scripts defining your desired state
- **Policies:** Governance rules and compliance policies integrated via Sentinel

Terraform Core

- The orchestration engine that reads templates, plans changes, manages state, and applies infrastructure modifications across your cloud providers seamlessly.

Provisioned Resources

- **Applications:** Deployed apps and containers
- **Storage:** Cloud storage buckets and volumes
- **Servers:** Virtual machines and compute instances
- **Networks:**VPCs, subnets, and firewalls

Terraform Providers & Resources

Providers

- Act as plugins that enable Terraform to communicate with APIs of cloud platforms or services.
- Each provider defines what resources and data sources it supports.

Common examples:

aws – Amazon Web Services

azurerm – Microsoft Azure

google – Google Cloud Platform

kubernetes – Kubernetes clusters

Example:

```
 provider "aws" {

region = "us-east-1"

          }
```

Declared using the **Provider** block in Terraform configuration.

Resources

- Represent real infrastructure objects such as EC2 instances, S3 buckets, or VPCs.
- Defined using a **resource** block with:

Type →	e.g., aws_instance
Name →	e.g., web
Arguments →	e.g., AMI ID, instance type, tags

- Terraform creates, updates, or destroys these resources based on the configuration.

Example:

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  tags = {
    Name = "WebServer"
  }
}
```

Terraform Init

Initializing Terraform – The Foundation Step

Command: Terraform init

Purpose: Prepares your working directory for Terraform usage

What it does:

- Downloads provider plugins (e.g., AWS, Azure, GCP)
- Initializes backend for state storage (local or remote)
- Validates configuration file structure

Typical usage: Run once when starting a new project or changing providers/backends.

Terraform Plan

Planning Infrastructure Changes

Command: Terraform plan

Purpose: Creates an execution plan showing what Terraform will do before making changes.

What it does:

- Compares the desired state (.tf files) vs current state (terraform.tfstate)
- Displays actions: Create, Update, or Destroy

Best Practices:

- Always review the plan before applying.

Terraform Apply

Applying Changes – Creating or Updating Resources

Command: Terraform apply

Purpose: Executes the actions proposed in the plan.

What it does:

- Calls APIs of cloud providers (via providers)
- Creates, updates, or deletes infrastructure as needed
- Updates terraform.tfstate file after successful execution

Terraform RPC (Remote Procedure Calls)

Terraform RPC – Behind-the-Scenes Communication

Key Points:

- RPC = Remote Procedure Call
- Used internally by Terraform CLI and its sub-processes to communicate.

Involves communication between:

- Terraform Core
- Provider plugins (like AWS, Azure, GCP)

Every provider runs as a separate process, and Terraform Core communicates via RPC.

Technical Overview:

- Terraform Core sends instructions to provider via RPC.
- Provider executes cloud API calls and sends responses back to Core.
- This architecture ensures plugin isolation, security, and performance.

Command	Purpose	Key Outcome
terraform init	Initialize working directory	Downloads providers, sets backend
terraform plan	Preview infrastructure changes	Shows add/update/destroy actions
terraform apply	Deploy changes	Creates/updates/deletes resources
RPC Mechanism	Internal communication	Enables provider-core interaction

Terraform Statefile (terraform.tfstate)

- Stores the mapping between Terraform configuration and real infrastructure resources.
- Acts as Terraform's **"source of truth"** for tracking the current state.
- Used by Terraform to determine what needs to be created, updated, or destroyed.
- Contains resource IDs, attributes, and dependencies.
- Should be stored securely – may include sensitive information.
- Can be stored remotely (e.g., S3, GCS, Azure Blob) for collaboration and state locking.

Command:

```
terraform state list
```

Terraform Destroy Command

Command: terraform destroy

Purpose:

- Deletes all resources tracked in the Terraform state file.
- Uses the state file to identify which resources to remove.
- Often used to clean up demo, test, or temporary environments.
- Interactive by default. Add -auto-approve to skip confirmation.

Example:

```
terraform destroy
terraform destroy -auto-approve
```

Terraform prevent_destroy

Purpose:

- The prevent_destroy setting is used to protect critical resources from being accidentally deleted by Terraform.
- When enabled, Terraform will block any destroy operation. Even from terraform destroy or a config change.
- Commonly used for prod resources

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

resource "aws_instance" "my_server" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"

  lifecycle {
    prevent_destroy = true
  }
}
```

Terraform Target Option

- The - target flag is used to apply or destroy specific resources instead of the entire configuration.
- Useful for incremental builds, testing, or partial updates.
- Helps when you want to isolate a single resource or module during troubleshooting.
- Should be used with caution — may cause dependency inconsistencies if used repeatedly.

Example

```
terraform destroy -target=aws_instance.web
terraform destroy -target=aws_instance.web -target=aws_s3.storage
```

Terraform fmt : Format Command

- The terraform fmt command automatically formats Terraform configuration files (.tf and .tfvars).
- Ensures consistent style, indentation, and alignment across all Terraform files.
- Makes your code clean, readable, and team-friendly.

Example:

```
terraform fmt              : Formats all .tf files in the current directory.
terraform fmt -recursive   : Format recursively (all subfolders)
```

terraform fmt -check -diff:

```
-check : Verifies if files are correctly formatted (doesn't change them).
-diff  : Shows differences between formatted and unformatted files.
```

Terraform Validate

- The terraform validate command is used to check the syntax and configuration correctness of your Terraform files.
- Ensures that all configuration files are valid and internally consistent before running plan or apply.
- Does not access any cloud provider APIs — purely a local static check.

Example:

```
terraform validate
```

Best Practice:

Run terraform fmt (to format code) before terraform validate for a clean workflow

Meta-arguments

In Terraform, meta-arguments are special arguments that you can use inside any resource, module, or data block. They are not specific to a particular provider or resource type—they work universally across Terraform configuration.

We have the following meta-arguments :

count, for_each : Create or manage multiple resources

depends_on : Enforce creation order

lifecycle : Add custom behavior rules

Provider : Use different provider configs

Terraform count Argument

Purpose:

- The count meta-argument is used to create multiple similar resources using a single resource block.
- Helps reduce code repetition and simplify configurations.
- You can reference each instance using its index value (count.index).
- Cannot create resources with different configurations -use 'for_each' instead for that.

Example

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

resource "aws_instance" "one" {
  count         = 3
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
  Name = “MyInstance-${count.index}”
}
}
```

Terraform for_each Argument

Purpose:

- For_each is a loop used to create multiple resources from a single resource block.
- Unlike count, it allows different configurations or unique identifiers per resource.
- Helps reduce repetitive code while maintaining flexibility.
- Each item in the list or map is assigned a unique key (each.key).
- Ideal for creating resources like EC2 instances, S3 buckets, or subnets with different names.

Example

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

resource "aws_instance" "one" {
  for_each = toset(["dev-server", "test-server", "prod-server"])
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"

  tags = {
    Name = each.key
  }
}
```

Terraform depends_on

Purpose:

- Used to manually define dependencies between resources.
- Ensures one resource is created or destroyed only after another resource is complete.
- Normally Terraform detects dependencies automatically, but depends_on helps when relationships aren’t direct.

Example

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

#Create an EC2 instance
resource "aws_instance" "my_server" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"
  tags = {
    Name = "Demo-Server"
  }
}

#Create an S3 bucket after EC2 is created
resource "aws_s3_bucket" "my_bucket" {
  bucket = "avinash-demo-bucket-12345"
  depends_on = [aws_instance.my_server]
}
```

Terraform Provider Argument

Purpose:

- It allows a resource to use a specific provider configuration when multiple provider blocks exist.
- Terraform allows multiple configurations of the same provider using the alias

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

provider "aws" {
  region = "ap-south-1"
  alias  = "india"
}

resource "aws_instance" "one" {
  ami           = "ami-020cba7c55df1f615"
  instance_type = "t2.micro"

  tags = {
    Name = "usa-server"
  }
}

resource "aws_instance" "two" {
  provider      = aws.india
  ami           = "ami-0f918f7e67a3323f0"
  instance_type = "t2.micro"

  tags = {
    Name = "india-server"
  }
}
```

Terraform lifecycle Block – Overview

- The lifecycle block defines special rules for how Terraform creates, updates, and destroys resources. It helps control the behavior of Terraform during changes to infrastructure.

Option	Description
create_before_destroy	Creates a new resource **before** destroying the old one.
prevent_destroy	Protects a resource from being destroyed accidentally.
ignore_changes	Ignores specific attributes during `terraform apply`.

Terraform lifecycle : create_before_destroy

Purpose:

- Ensures Terraform creates a new resource first, and destroys the old one later - preventing downtime.
- Commonly used for servers, load balancers, or databases in production.
- When a change requires replacement (e.g., new AMI or name change),
- Terraform first creates the new instance, then destroys the old one after the new one is ready.

Example:

```
resource "aws_instance" "my_server" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"

  lifecycle {
    create_before_destroy = true
  }
}
```

Terraform prevent_destroy

Purpose:

- The prevent_destroy setting is used to protect critical resources from being accidentally deleted by Terraform.
- When enabled, Terraform will block any destroy operation. Even from terraform destroy or a config change.
- Commonly used for prod resources

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

resource "aws_instance" "my_server" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"

  lifecycle {
    prevent_destroy = true
  }
}
```

Terraform lifecycle : ignore_changes

Purpose:

- Tells Terraform to ignore specific attribute updates during terraform apply.
- Useful when certain attributes are changed manually outside Terraform.
- Terraform will not try to revert those changes.

Example:

```
resource "aws_instance" "my_server" {
  ami           = "ami-046d18c147c36bef1"
  instance_type = "t2.micro"

  lifecycle {
    ignore_changes = [instance_type]
  }
}
```

Default Parallelism

- By default, Terraform performs up to 10 concurrent operations (in parallel).
- That's where the confusion often comes from-not a limit on total resources, but on simultaneous executions.
- Terraform can create hundreds or even thousands of resources in a single
- terraform apply. it just does them in batches of 10 by default.

terraform apply -parallelism=20 : increases the parallelism to 20

Terraform replace (Formerly taint)

- Terraform replace is a Terraform command that forces a resource to be destroyed and recreated on the next terraform apply — even if there is no actual configuration change.
- Equivalent to the old deprecated command: terraform taint <resource>

It is used when:

- A resource is corrupted
- Manual changes were made outside Terraform
- You want a clean recreation without modifying the code
- You want to simulate taint behavior

```
terraform plan -replace="aws_instance.mumbai-Instance"
terraform apply -replace="aws_instance.my_server"
```

Terraform refresh - deprecated

- Terraform refresh was a Terraform command used to update the state file with the actual real-world infrastructure state—without making any changes to the infrastructure itself.
- Look at the current real cloud state (AWS/GCP/Azure)
- Compare it with the Terraform state file
- Update the state file to match the real infrastructure
- WITHOUT updating your .tf configuration
- WITHOUT making any actual changes to cloud resources

In simple words: It overwrote your state file with whatever is currently in the cloud.
As of Terraform 1.0+, terraform refresh is deprecated.

What is Drift?

Purpose:

- Drift happens when the real infrastructure changes outside of Terraform —meaning the infra no longer matches what Terraform expects in the state file

Example:

- You manually stopped an EC2 instance
- Someone changed an SG rule
- A tag was added/removed
- Instance type changed in AWS console

Drift is detected during:

- terraform plan
- terraform apply
- automatic refresh (because refresh is built into plan/apply)

Drift Detection Steps (Internally):

Read Terraform state file

- Query provider APIs for actual resource values (e.g., AWS DescribeInstances, DescribeSecurity Groups, etc.)
- Compare actual values with state values
- Mark differences as drift
- Display drift in the plan output

Plan (summary — do these steps)

- Create a simple EC2 using Terraform (t2.micro).
- Confirm it's created.
- Manually change the instance type in AWS to t3.micro. This causes drift.
- Run terraform plan or "terraform plan -refresh-only" to see the drift.

Fix: either update Terraform code and apply, or run terraform apply to reconcile/restore.

Terraform Import

Purpose:

terraform import allows Terraform to take control of an existing resource created outside Terraform.
It imports the real resource into the terraform.tfstate file.
After importing, terraform plan works normally.

Before importing, you must have:
- A resource block written in .tf
- The resource ID from AWS

After import → run terraform plan to check for differences.

```
terraform init
terraform plan -generate-config-out=ec2.tf
terraform apply
terraform destroy (to verify the output)
```

Example:

```
provider "aws" {
  region = "ap-south-1"
}

resource "aws_instance" "myinstance" {
  ami           = "ami-03695d52f0d883f65"
  instance_type = "t3.micro"
}

import {
  to = aws_instance.myinstance
  id = "i-07ccd22283d28c3c7"
}
```

Terraform Debugging Options

1. TF_LOG Environment Variable

Enables verbose logs from Terraform.

Levels:

- TRACE – Most detailed
- DEBUG – Debug-level info
- INFO – General information
- WARN – Warnings
- ERROR – Errors only

Usage:

```
export TF_LOG=TRACE
terraform plan

unset TF_LOG
```

2. TF_LOG_PATH

Creates a detailed log file for later analysis.

```
export TF_LOG=DEBUG
export TF_LOG_PATH="terraform.log"
terraform apply
```

terraform plan -debug

Runs Terraform with detailed debugging information using below command

Terraform Statefile (terraform.tfstate)

- Stores the mapping between Terraform configuration and real infrastructure resources.
- Acts as Terraform's **"source of truth"** for tracking the current state.
- Used by Terraform to determine what needs to be created, updated, or destroyed.
- Contains resource IDs, attributes, and dependencies.
- Should be stored securely – may include sensitive information.
- Can be stored remotely (e.g., S3, GCS, Azure Blob) for collaboration and state locking.

Command:

```
terraform state list
```

Terraform State

- Terraform refreshes the state automatically during plan, apply, and destroy.
- A backend defines where and how the state file is stored and managed.
- By default, Terraform uses the local backend, storing terraform.tfstate in the project folder in JSON format.
- When changes occur or resources are deleted, Terraform creates a backup file named terraform.tfstate.backup.

Why State File Locking Is Important

- Prevents simultaneous changes by allowing only one Terraform operation at a time.
- Avoids conflicts such as duplicate resources or unintended deletions.
- Automatically handled by backends that support locking.
- Not supported by all backends — the local backend has no locking, while most remote backends do.

Terraform Backend Block

- By default, Terraform has no backend configuration block, so it uses the local backend.
- This is why the terraform.tfstate file is stored in the current working directory.
- You can define a backend block to store the state file in a different location.

```
terraform {
 backend "local" {
  path = "/tmp/terraform.tfstate"
 }
}
```

Terraform S3 Backend

- The default local backend stores the state file on your machine, limiting collaboration.
- Real-world teams need a centralized and consistent storage location for the Terraform state file.
- Amazon S3 is a popular remote backend choice for securely storing and sharing the state file.
- Using an S3 backend enables team collaboration, better reliability, and centralized state management.

Example:

```
terraform {
  backend "s3" {
    bucket = "aviz-bucket"
    key    = "path/to/my/key"
    region = "ap-south-1"
  }
}
```

Moving Terraform State: Local ↔ S3 Backend

- "terraform init -migrate-state" is used to move the existing Terraform state from one backend to another.
- **Common scenario:** moving from local state → S3 backend or back to local.
- Terraform init -migrate-state

Example:

```
terraform {
  backend "s3" {
    bucket = "aviz-bucket"
    key     = "path/to/my/key"
    region = "ap-south-1"
   use_lockfile = true
  }
}
```

Why Does State Locking Happen?

➦ **Scenario:**

Two engineers -Person A and Person B -both run terraform apply at the same time on the same project.

➦ Person A starts an apply → Terraform begins updating the state.

➦ Before it finishes, Person B also triggers an apply.

Without locking:

➦ Both operations modify the state at the same time.

This can lead to:

- Duplicate resources
- Wrong resource deletions
- Corrupted state file
- Unpredictable infrastructure changes

With locking:

- When Person A starts, the state file is locked.
- Person B must wait until the lock is released.
- Ensures safe, consistent, and predictable updates.

Terraform state commands

- Instead of editing the state file manually, Terraform provides safe state subcommands.

List resources	: terraform state list
Show details of a specific resource	: terraform state show aws_instance.myinstance
Remove a resource from state	: terraform state rm aws_instance.myinstance
Rename a resource in the state	: terraform state mv aws_instance.one aws_instance.onee

What is a Terraform Provider?

- A Terraform provider is a plugin that allows Terraform to interact with external platforms and services (e.g., AWS, Azure, GitHub).
- Providers expose resources and data sources, enabling Terraform to create, read, update, and delete infrastructure.
- When you run `terraform init`, Terraform automatically downloads the latest compatible provider plugins.
- Some Terraform configurations may not work with older provider versions, making timely updates necessary.
- You can view the most recent provider releases in the Terraform Registry: (Terraform Registry → Providers).
- Whenever you introduce a new provider in your configuration, running `terraform init` is mandatory to download and initialize it.

Why Do We Need `required_providers`?

- Defines which provider Terraform must download
- Ensures consistent provider version across:
  - Local machine
  - Teammates
  - CI/CD pipelines
- Prevents unexpected breakages due to auto-upgraded providers
- Required syntax in modern Terraform (0.13+)

1. Official Providers — Developed and maintained by HashiCorp
2. Partner Providers — Managed by trusted third-party organizations
3. Community Providers — Created and maintained by individual contributors

Role of Provider Block vs Required Providers Block

- **Provider Block (`provider "aws" {}`)**
- Configures how to connect to AWS
  Example settings:
  - Region
  - Profile
  - Credentials
  - Used at runtime
- **Required Providers Block (`terraform { required_providers {} }`)**
- Defines which provider and what version to use
- Controls plugin download
- Provides version stability

Risks of Not Using or Updating Providers

- Auto-updates can break your code unexpectedly
- Terraform plans become inconsistent across team
- Incompatibility with latest Terraform releases
- Failure to access newly introduced AWS services
- Deprecated fields may cause apply errors later

Significance of Upgrading Provider Versions

- Unlocks new AWS features (new services, updated APIs)
- Includes bug fixes & performance improvements
- Fixes security issues or vulnerabilities
- Removes old/deprecated fields → keeps code modern
- Ensures compatibility with new Terraform versions

Provider Version Locking?

Provider version locking ensures:

- Consistent provider versions across all environments
- Predictable and stable infrastructure behavior
- Prevention of unexpected breaking changes caused by automatic upgrades
- You lock a provider version using the `required_providers` block:
- Terraform will always use the specified version unless you explicitly upgrade it.v

Example:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "6.21.0"
    }
  }
}
```

Terraform variables

- Variables allow you to supply values to your configuration without modifying the main Terraform code.
- They enable dynamic configuration and help maintain clean, reusable, and scalable infrastructure code.

Benefits of Using Terraform Variables

- Reusability: The same Terraform configuration can be reused across multiple environments (Dev, Test, Prod) simply by providing different variable values.
- Clean: No hardcoded values; configuration stays readable and maintainable.
- Flexible: Values can be changed at runtime via CLI, tfvars files, or environment variables.
- Consistent: Centralized variables ensure uniform settings across resources.
- Secure (when marked sensitive): Keeps secrets out of the main code.

Type	Description	Example
String	Text value	"dev", "us-east-1"
Number	Numeric value	10, 500
Bool	True or False	true, false
List	Ordered collection of values	["t2.micro", "t3.micro"]
Map	Key-value pairs	{env = "dev", tier = "1}
Object	Group of named attributes	{name = "dev", size = 2}
Tuple	Ordered list with different data types	[" dev", 2, true]

String

```
provider "aws" {
  region = "ap-south-1"
}

variable "instance_type" {
  type    = string
  default = "t2.micro"
}

resource "aws_instance" "example_string" {
  ami           = "ami-03695d52f0d883f65"
  instance_type = var.instance_type
}
```

List

```
provider "aws" {
  region = "ap-south-1"
}

variable "instance_types" {
  type    = list(string)
  default = ["t2.micro", "t3.micro", "t3.small"]
}

resource "aws_instance" "example_list" {
  ami           = "ami-03695d52f0d883f65"
  instance_type = var.instance_types[1]   # picks "t3.micro"
}
```

Count

```
provider "aws" {
  region = "ap-south-1"
}
variable "instance_count" {
  type    = number
  default = 3
}
resource "aws_instance" "example_count" {
  count         = var.instance_count
  ami           = "ami-03695d52f0d883f65"
  instance_type = "t2.micro"
  tags = {
    Name = "server-${count.index}"
  }
}
```

Map

```
provider "aws" {
  region = "ap-south-1"
}
variable "instance_tags" {
  type = map(string)
  default = {
    Name = "map-example"
    Env  = "dev"
    App  = "demo-app"
  }
}
resource "aws_instance" "example_map" {
  ami           = "ami-03695d52f0d883f65"
  instance_type = "t2.micro"
  tags = var.instance_tags
}
```

Terraform Modules

- Terraform modules help you organize infrastructure code into a clean, scalable folder structure. A module is essentially a collection of related resources that are managed together. Using modules improves readability, reduces duplication, and enables reusability across teams or environments.
- Modules group multiple resources into logical units that can be reused and shared across your organization.
- They promote consistency and reduce errors by encapsulating best-practice configurations.
- Modules can be published privately or publicly for others to consume.
- Each module operates independently and downloads the required provider plugins as needed.
- Specifying a module version is recommended for production, but not strictly mandatory.
- Downloaded module code is stored under the .terraform/modules/ directory.

Types of Modules

Root Module:

- The top-level directory where Terraform commands are executed. All .tf files in this directory collectively form the root module.

- **Child Modules:**

  Modules called or referenced by the root module or other modules. They help break down complex infrastructure into smaller, maintainable components.

Simple Project structure

```
project /
|
├── main.tf          # Root module
├── variables.tf      # Root variables (optional)
|
└── modules/
    ├── ec2/
    |   └── main.tf     # EC2 child module
    |
    └── s3/
        └── main.tf     # S3 child module
```

Child Module: EC2 (modules/ec2/main.tf)

```
resource "aws_instance" "this" {
  ami           = "ami-03695d52f0d883f65"
  instance_type = "t3.micro"
}
```

Child Module: S3 (modules/s3/main.tf)

```
resource "aws_s3_bucket" "this" {
  bucket = "my-test-bucket-12345"
}
```

Root Module: main.tf

```
provider "aws" {
  region = "ap-south-1"
}

module "ec2_instance" {
  source = "./modules/ec2"
}

module "s3_bucket" {
  source = "./modules/s3"
}
```

Terraform Module Sources

1. Local path
```
module "ec2" {
  source = "./modules/ec2"
}
```

2. Terraform Registry
```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.5.0"
}
```

3. Git Repository
```
module "vpc" {
  source = "git::https://github.com/avizway/aws-vpc-module.git"
}
```

4. Private Registry (Enterprise)
```
module "network" {
  source  = "app.terraform.io/mycompany/network/aws"
  version = "1.2.0"
}
```