Terraform

Terraform

- Terraform is an open source infrastructure as code (IaC) tool created by HashiCorp.
- It allows users to easily define and provision compute, network, and storage resources on public and private cloud platforms, as well as on-premises data centers.

What Is Infrastructure as Code?

- Infrastructure as code (IAC) lets you write and execute code to define, deploy, update, and destroy your infrastructure.

There are five broad categories of IAC tools:

1. Ad hoc scripts [For example, a Bash script that configures a web server by installing dependencies, checking out some code from a Git repo, and firing up an Apache web server]
2. Configuration management tools [Chef, Puppet, Ansible, and SaltStack are all configuration management tools,which means that they are designed to install and manage software on
   existing servers]
3. Server templating tools [AMI's of AWS]
4. Orchestration tools [Kubernetes for manging the cluster]
5. Provisioning tools [Terraform]

The Benefits of Infrastructure as Code:

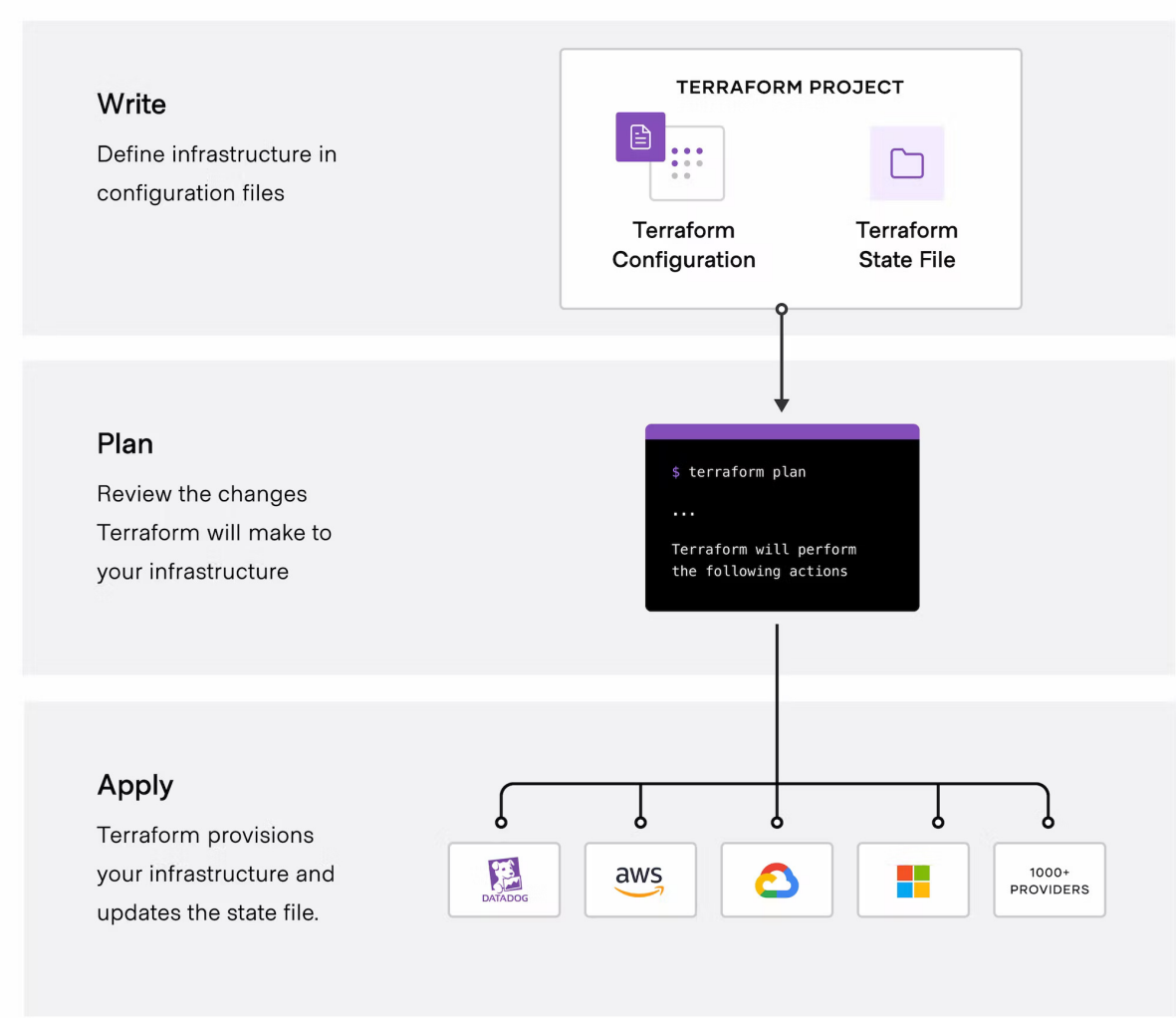
1. Self-service
2. Speed and safety
3. Documentation
4. Version control
5. Validation
6. Reuse

How does Terraform work?

→ Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs). Providers enable Terraform to work with virtually any platform or service with an accessible API.

→ The core Terraform workflow consists of three stages:

- **Write:** You define resources, which may be across multiple cloud providers and services. For example, you might create a configuration to deploy an application on virtual machines in a Virtual Private Cloud (VPC) network with security groups and a load balancer.
- **Plan:** Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
- **Apply:** On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies. For example, if you update the properties of a VPC and change the number of virtual machines in that VPC, Terraform will recreate the VPC before scaling the virtual machines.



Build Infrastructure:

```
terraform {
  required_providers {
    aws = {
        source = "hashicorp/aws"
        version = "~> 4.16"

    }
  }
}

provider "aws" {
    region = "us-east-1"  //Name of the AWS Region
}

resource "aws_instance" "MyExampleInstance" {
  ami = "ami-0557a15b87f6559cf"  //AMI image name
  instance_type = "t2.micro"  //type of the instance to be provisioned
}
```

Sequence of Usage Commands:

- terraform init → To tell Terraform to scan the code, figure out which providers you're using, and download the code for them. By default, the provider code will be downloaded into a .terraform
  folder, which is Terraform's scratch directory.
- terraform plan → The plan command lets you see what Terraform will do before actually making any changes. The output of the plan command is similar to the output of the diff command that is part of Unix, Linux, and git: anything with a plus sign (+) will be created, anything with a minus sign (–) will be deleted, and anything with a tilde sign (~) will be modified in place.
- terraform apply → The apply command shows you the same plan output and asks you to confirm whether you actually want to proceed with this plan. So, while plan is available as a separate command, it's mainly useful for quick sanity checks and during code reviews and most of the time you'll run apply directly and review the plan output it shows you.
- terraform destroy → The destroy command destroys the environment that was created.

What Is Terraform State:

→ Every time you run Terraform, it records information about what infrastructure it created in a Terraform state file.

→ This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world.

→ Using this JSON format, Terraform knows that a resource with type aws_instance and name example corresponds to an EC2 Instance in your AWS account with ID i-00d689a0acc43af0f.

→ The output of the plan command is a diff between the code on your computer and the infrastructure deployed in the real world, as discovered via IDs in the state file.

> THE STATE FILE IS A PRIVATE API

→ If you are using terraform for your personal project then the state file lives locally and it works fine, but when your working on a real time project then, there are several issues:

- Shared storage for state files → To be able to use Terraform to update your infrastructure, each of your team members needs access to the same Terraform state files. That means
  you need to store those files in a shared location.
- Locking state files → As soon as data is shared, you run into a new problem: locking. Without locking, if two team members are running Terraform at the same time,
  you can run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.
- Isolating state files → When making changes to your infrastructure, it's a best practice to isolate different environments. For example, when making a change in a testing or staging environment, you want to be sure that there is no way you can accidentally break production.

→ Remote backends solve all three of the issues just listed 🙂