1. Terraform :

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

Terraform is built on a plugin-based architecture

1. **Terrraform Commands:**

Common commands:

apply Builds or changes infrastructure

console Interactive console for Terraform interpolations

destroy Destroy Terraform-managed infrastructure

fmt Rewrites config files to canonical format

get Download and install modules for the configuration

graph Create a visual graph of Terraform resources

import Import existing infrastructure into Terraform

init Initialize a new or existing Terraform configuration

output Read an output from a state file

plan Generate and show an execution plan

providers Prints a tree of the providers used in the configuration

push Upload this Terraform module to Terraform Enterprise to run

refresh Update local state file against real resources

show Inspect Terraform state or plan

taint Manually mark a resource for recreation

untaint Manually unmark a resource as tainted

validate Validates the Terraform files

version Prints the Terraform version

workspace Workspace management

1. **Terraform**

Terraform must store state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

This state is stored by default in a local file named "terraform.tfstate", but it can also be stored remotely, which works better in a team environment.

Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a [refresh](https://www.terraform.io/docs/commands/refresh.html) to update the state with the real infrastructure.

Benefit of Backend :

**Working in a team**

Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Enterprise even automatically store a history of all state revisions.

**Keeping sensitive information off disk**

State is retrieved from backends on demand and only stored in memory. If you're using a backend such as Amazon S3, the only location the state ever is persisted is in S3

**Remote operations**

For larger infrastructures or certain changes, terraform apply can take a long, long time. Some backends support remote operations which enable the operation to execute remotely. You can then turn off your computer and your operation will still complete. Paired with remote state storage and locking above, this also helps in team environments.

Backend Types

* **Standard**: State management, functionality covered in [State Storage & Locking](https://www.terraform.io/docs/backends/state.html)

Artifactory,,azurerm,consul,etcd,etcdv3

gcs

http

manta

s3

swift

terraform enterprise

* **Enhanced**: Everything in standard plus [remote operations](https://www.terraform.io/docs/backends/operations.html).

**Local/remote**

1. Terraform S3 Backend Configuration

terraform {

backend "s3" {

bucket = "digcore-deploy"

key = "pim/prod/terraform.tfstate"

region = "eu-west-1"

profile = "asos-digcore"

}

1. Terraform Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction. Provisioners can be used to bootstrap a resource, cleanup before destroy, run configuration management, etc.

Provisioners are added directly to any resource:

resource "aws\_instance" "web" {

# ...

provisioner "local-exec" {

command **=** "echo ${self.private\_ip} > file.txt"

}

}

Some of the provisioners that are available in terraform

**File, local-exec, remote-exec, connection, chef**

1. Terraform taint and untaint

The **terraform taint** command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply.

This command will not modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next [plan](https://www.terraform.io/docs/commands/plan.html) will show that the resource will be destroyed and recreated and the next[apply](https://www.terraform.io/docs/commands/apply.html) will implement this change.

The **terraform untaint** command manually unmarks a Terraform-managed resource as tainted, restoring it as the primary instance in the state. This reverses either a manual terraform taint or the result of provisioners failing on a resource.

This command will not modify infrastructure, but does modify the state file in order to unmark a resource as tainted.

1. Variables
2. terraform apply -var-file**=**foo.tfvars -var-file**=**bar.tfvars
3. terraform apply -var 'foo={quux="bar"}' -var 'foo={bar="baz"}'

Interpolation:

${var.foo} 🡺 String

${var.amis["us-east-1"]} 🡺 Map

${var.subnets[idx]} 🡺 list

${self.private\_ip} 🡺 attributes of own resources

8. Data Sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to build on information defined outside of Terraform, or defined by another separate Terraform configuration