# Terraform

Terraform

Raman Khanna

# Introduction

Your  Name

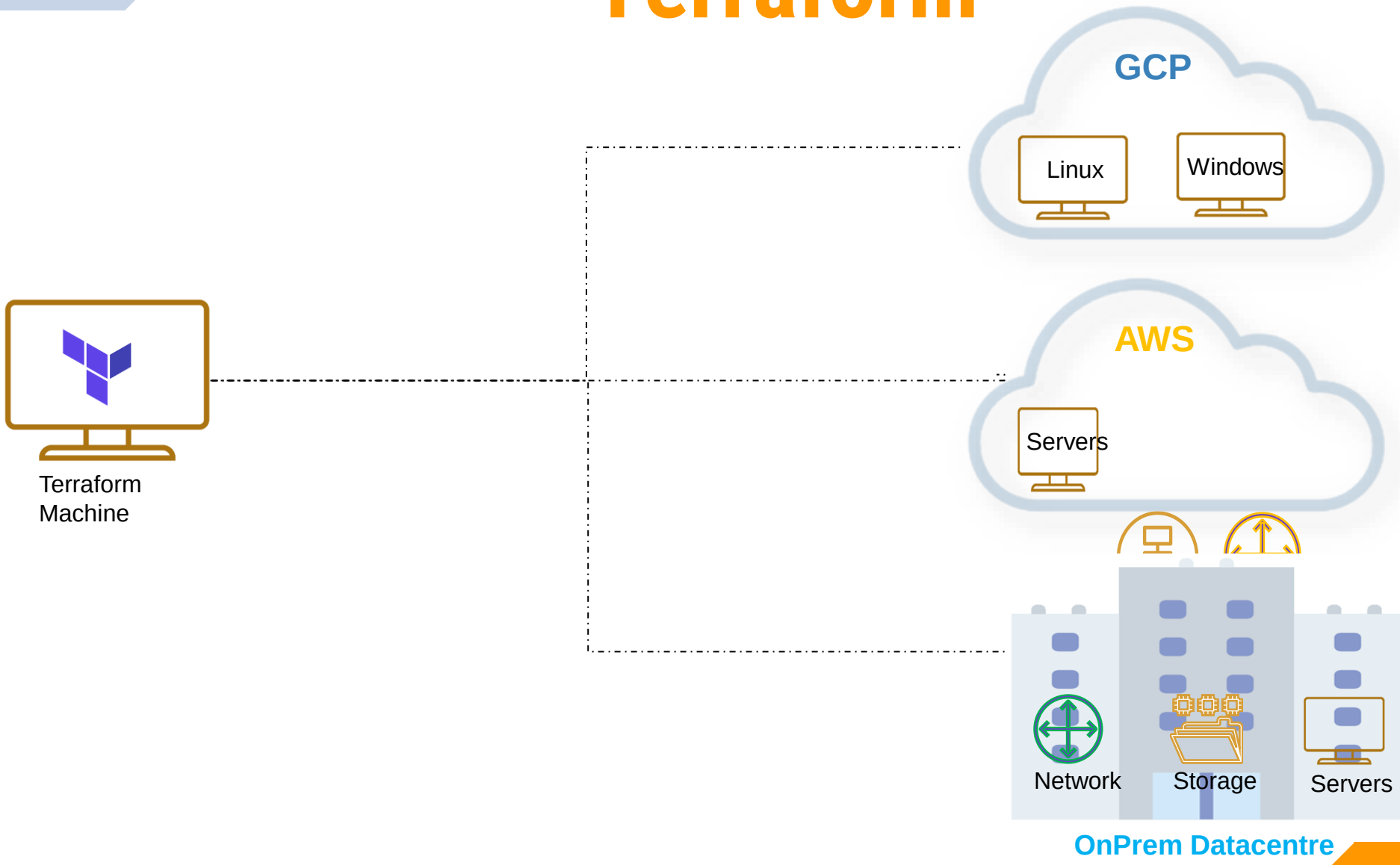
Total experience

Background – Development / Infrastructure / Database / Network

Experience on AWS Cloud and Terraform

# What is Orchestration?

# Terraform


GCP
Linux
Windows
AWS
Servers
Terraform
Machine
Network
Storage
Servers
OnPrem Datacentre

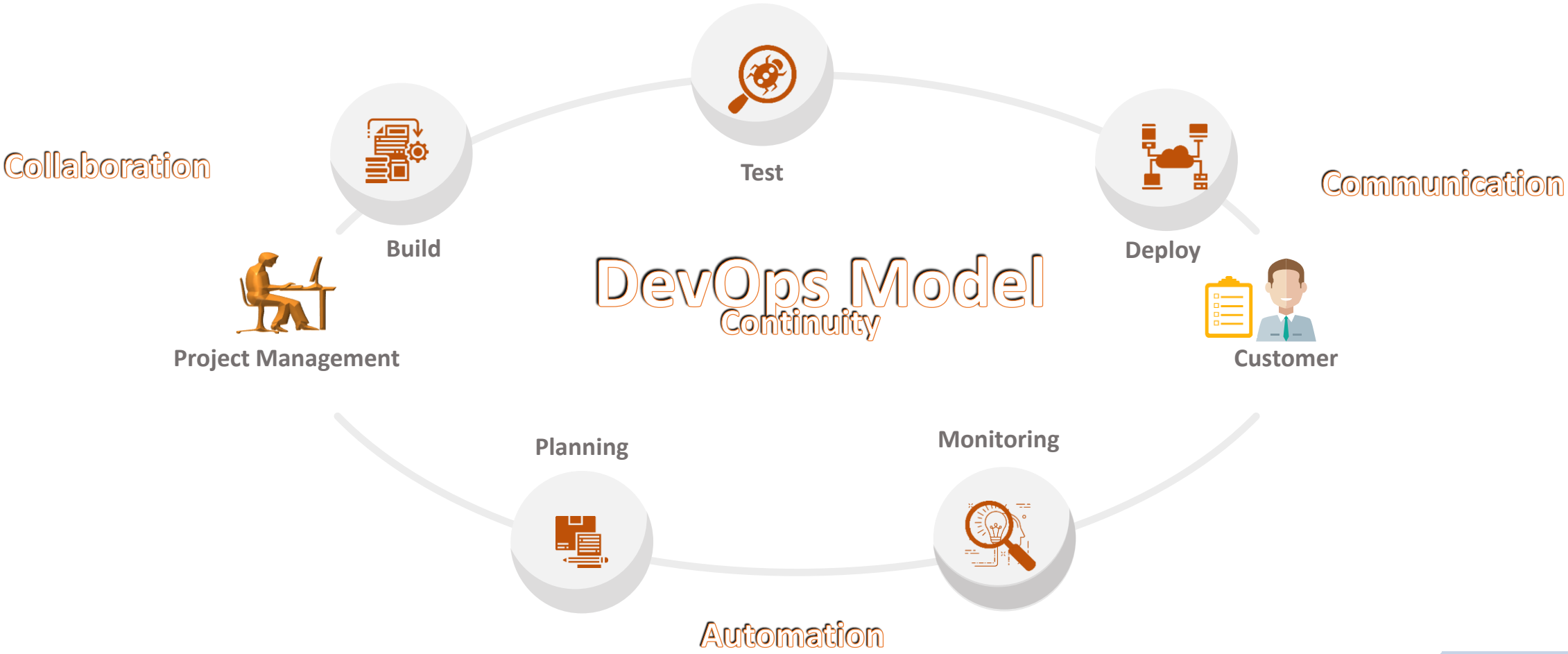
# GUI vs CLI vs IAC

- **GUI (Graphical User Interface)**
  - ✓ Best for end user experience
  - ✓ Easy management
  - ✓ Bad for Automation
  - ✓ Not helpful for Administrators

- **CLI (Command Line Interface)**
  - Best for Admin Experience
  - Easy management for Admin level tasks
  - Bad for end user experience
  - Bad for maintaining desired state and consistency

- **IaC (Infrastructure as Code)**
  - Best for Admin Experience
  - Easy management for Admin tasks
  - Easy to understand for end users too
  - Can easily maintain consistency and desired state
  - Infrastructure is written in files, so can be versioned

# DevOps


Collaboration
Build
Test
Deploy
Communication
Project Management
DevOps Model
Continuity
Customer
Planning
Monitoring
Automation

# DevOps in Action

Continuous Feedback

Continuous Improvement

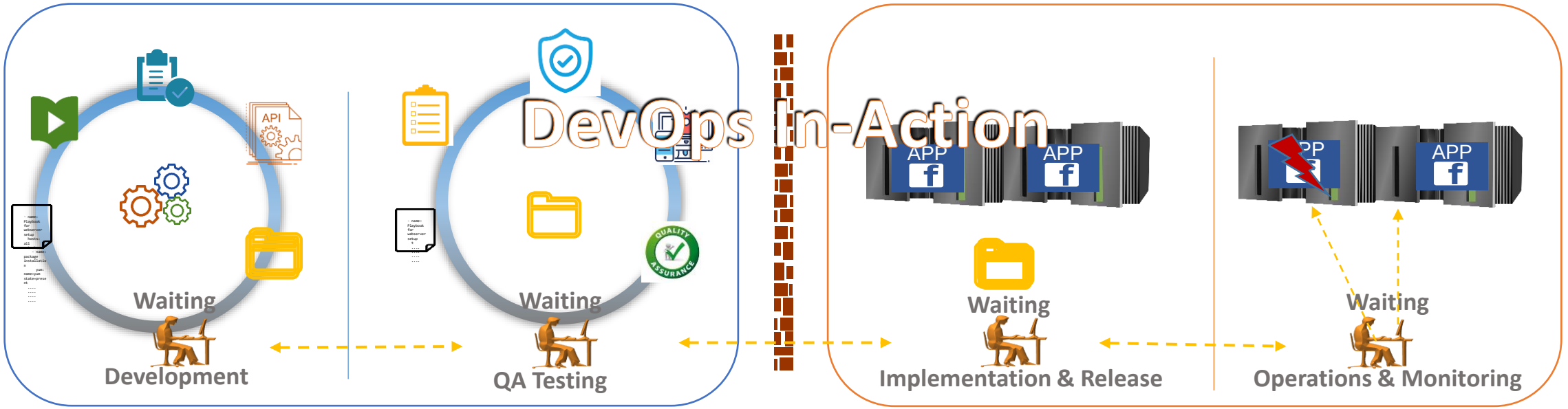
Continuous Planning

Continuous Integration

Continuous Delivery

Continuous Deployment

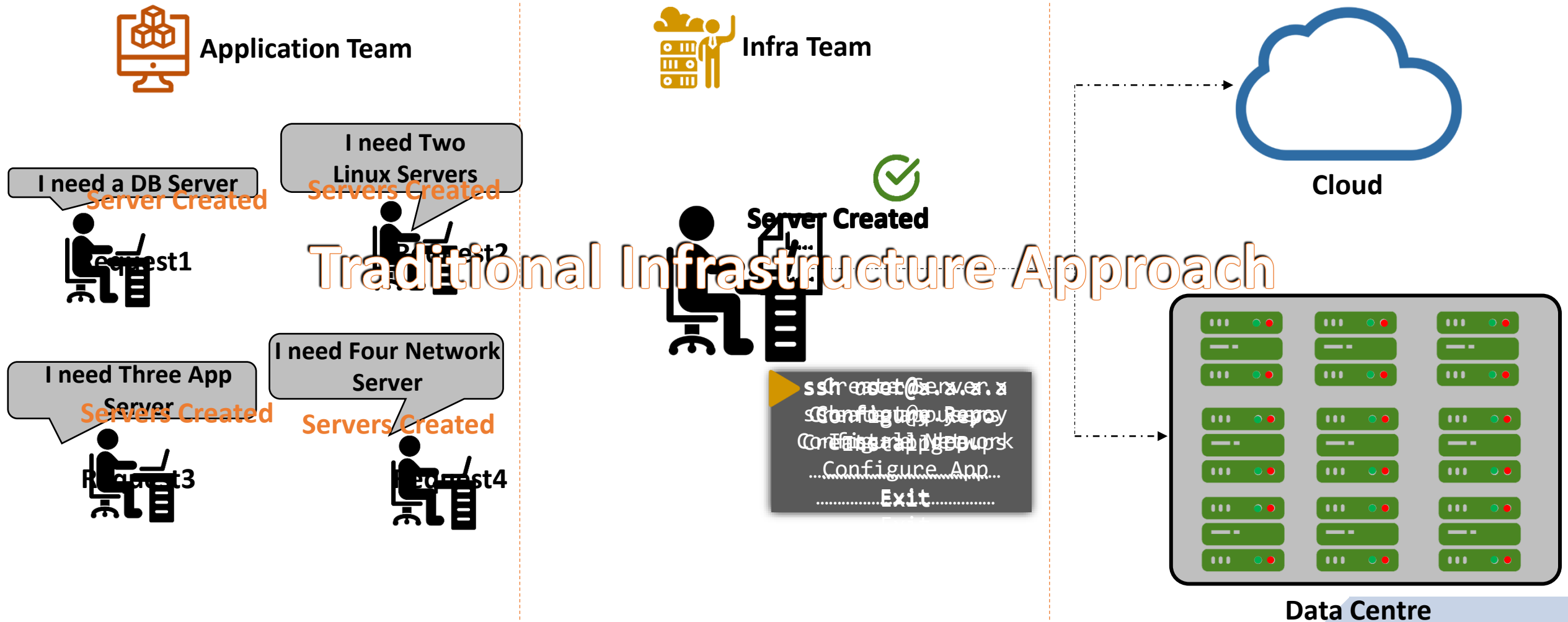
Continuous Monitoring




Development

Operations

# Why DevOps IaC




Traditional Infrastructure Approach

# IAC (Infrastructure as Code)

Application Team

I need DB Server

Server Created

Request1

I need Two Linux Servers

Servers Created

Request2

I need Three Linux Servers

Servers Created

Request3

I need Four Linux Servers

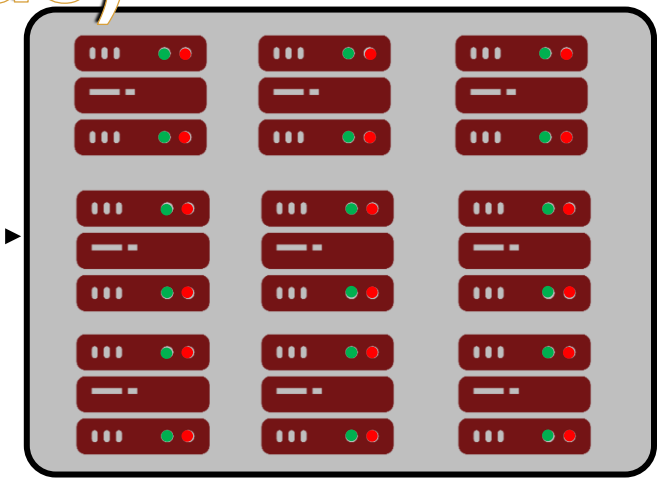
Servers Created

Request4

Infra Team

```
File is: main.tf
provider "aws" {
  region = "us-east-1"
}
resource "aws_instance" “requestfour" {
  count = "4"
  ami = "ami-03d5c68bab01f3496"
  instance_type = "t2.micro"
  tags = {
    Name = "DevOpsInAction"
  }
}
output "myawsserver" {
  value =
"${aws_instance.myawsserver.public_ip}"
}
```

Cloud



Data Centre

IaC is Managing Infrastructure in files rather than manually configuring resources in a user interface

# Terraform

Terraform is an easy-to-use IT Orchestration & Automation Software for System Administrators & DevOps Engineers.

- It is the infrastructure as code offering from Hashicorp.
- It is a tool for building, changing, and managing infrastructure in a safe, repeatable way.
- Configuration language called the HashiCorp Configuration Language (HCL) is used to configure the Infrastructure.
- Compatible with almost all major public and private Cloud service provider

# Terraform

Infrastructure as code (IAC)

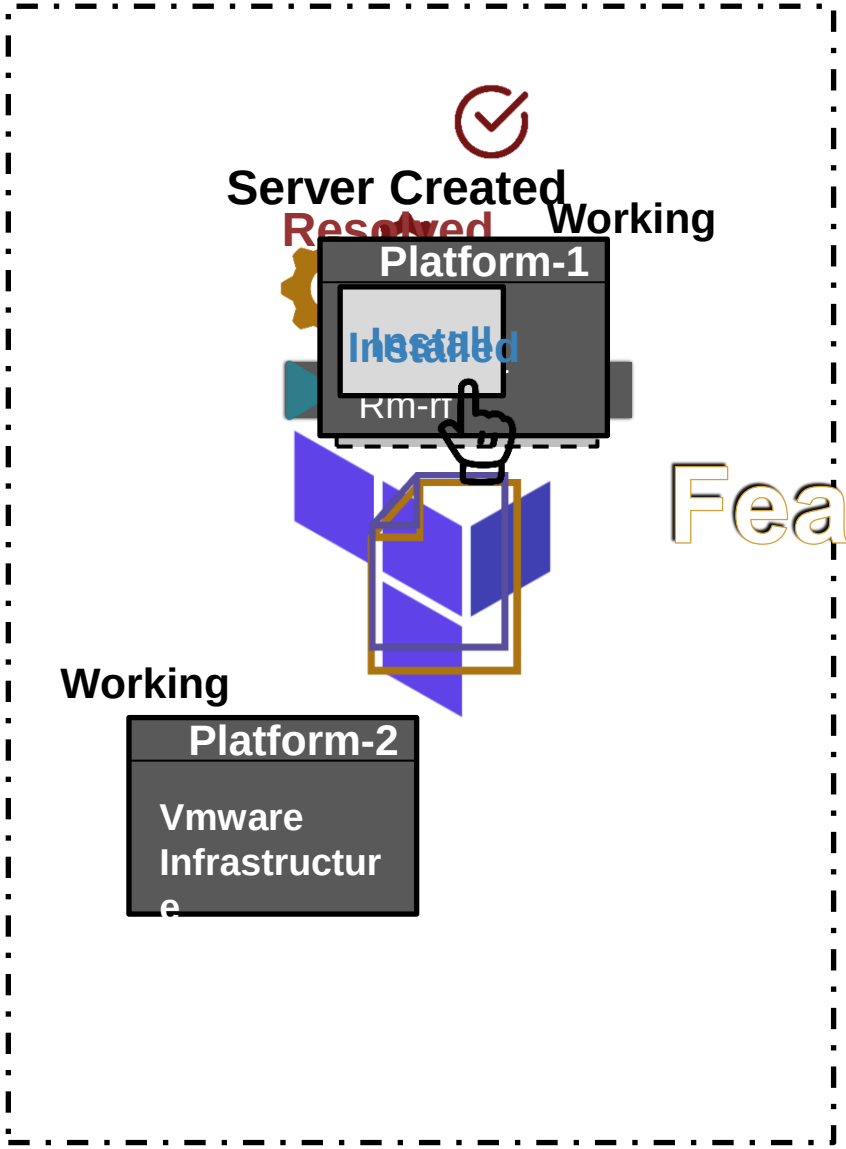

July 2014, HashiCorp

## What is Terraform?

Opensource / Enterprise

HCL (Hashicorp Configuration Language)

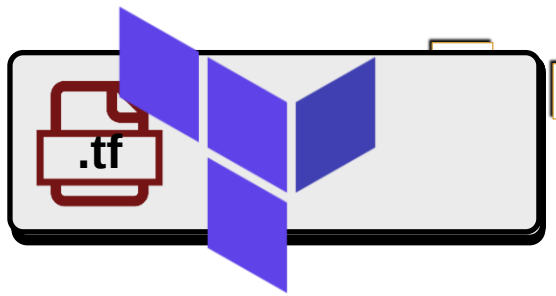
# Terraform




## Feature & Advantages

- Easy Installation
- Declarative in Nature
- Idempotent
- Intelligent Dependency Resolver
- Platform Agnostic
- Simple and easy to use

# Terraform




## Terraform Terminologies

**Providers**

**Variables**

**Resources**

**Provisioners**

**DataSources**

**Outputs**

**Modules**

**File extension**
**.tf**

# Terraform

main.tf

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

Provider Block

```
resource "aws_instance" "myserver" {
  ami = "ami-030ff268bd7b4e8b5"
  instance_type = "t2.micro"
  tags = {
    Name = "DevOpsInAction"
  }
}
```

Resource Block

```
output "myserveroutputs" {
  description = "Display Servers Public IP"
  value = "${aws_instance.myserver.public_ip}"
}
```

Output Block

## Terraform File (Sample Code)

# Why Terraform?

- Infrastructure as Code – Write stuff in files, Version it, share it and collaborate with team on same.
- Declarative in Nature
- Automated provisioning
- Clearly mapped Resource Dependencies
- Can plan before you apply
- Consistent
- Compatible with multiple providers and infra can be combined on multiple providers
- 50+ list of official and verified providers
- Approx. 2500+ Modules readily available to work with
- Both Community and Enterprise versions available
- A best fit in DevOps IaC model

# Why Terraform?

- **Platform Agnostic** – Manage Heterogeneous Environment
- **Perfect State Management** – Maintains the state and Refreshes the state before each apply action.

  Terraform state is the source of truth. If a change is made or a resource is appended to a configuration, Terraform compares those changes with the state file to determine what changes result in a new resource or resource modifications.
- **Confidence**: Due to easily repeatable operations and a planning phase to allow users to ensure the actions taken by Terraform will not cause disruption in their environment.

# Terraform and its Peers

- Chef
- Puppet
- SaltStack
- Ansible
- CloudFormation
- Terraform
- Kubernetes

# Terraform and its Peers

Many tools available in Market. Few things to consider, before selecting any tool:

- Configuration Management vs Orchestration
- Mutable Infrastructure vs Immutable Infrastructure
- Procedural vs Declarative

# Terraform and its Peers

| | Chef | Puppet | Ansible | SaltStack | CloudFormation | Terraform |
|---|---|---|---|---|---|---|
| **Code** | Open source | Open source | Open source | Open source | Closed source | Open source |
| **Cloud** | All | All | All | All | AWS only | All |
| **Type** | Config Mgmt | Config Mgmt | Config Mgmt | Config Mgmt | Orchestration | Orchestration |
| **Infrastructure** | Mutable | Mutable | Mutable | Mutable | Immutable | Immutable |
| **Language** | Procedural | Declarative | Declarative | Declarative | Declarative | Declarative |
| **Architecture** | Client/Server | Client/Server | Client-Only | Client/Server | Client-Only | Client-Only |

# Knowledge Checks

- What is Configuration Management?
- What is Orchestration?
- List a few available configuration Management tools.
- What are the Advantages of Terraform?

# Summary: Terraform

Terraform is an easy-to-use IT Orchestration & Automation, Software for System Administrators & DevOps Engineers.

- 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.
- Maintain Desired State
- Highly scalable and can create a complete datacenters in minutes
- Agentless solution
- Declaration in nature than Procedural
- Uses Providers API to provision the Infrastructure
- Terraform creates a dependency graph to determine the correct order of operations.

# AWS

# Amazon Web Services

AWS (Amazon Web Services) is a group of web services (also known as cloud services) being provided by Amazon since 2006.

AWS provides huge list of services starting from basic IT infrastructure like CPU, Storage as a service, to advance services like Database as a service, Serverless applications, IOT,  Machine Learning services etc..

Hundreds of instances can be build and use in few minutes as and when required, which saves ample amount of hardware cost for any organizations and make them efficient to focus on their core business areas.

Currently AWS is present and providing cloud services in more than 190 countries.

Well-known for IaaS, but now growing fast in PaaS and SaaS.

# Why AWS?

**Low Cost:** AWS offers, pay as you go pricing. AWS models are usually cheapest among other service providers in the market.

**Instant Elasticity:** You need 1 server or 1000's of servers, AWS has a massive infrastructure at backend to serve almost any kind of infrastructure demands, with pay for what you use policy.

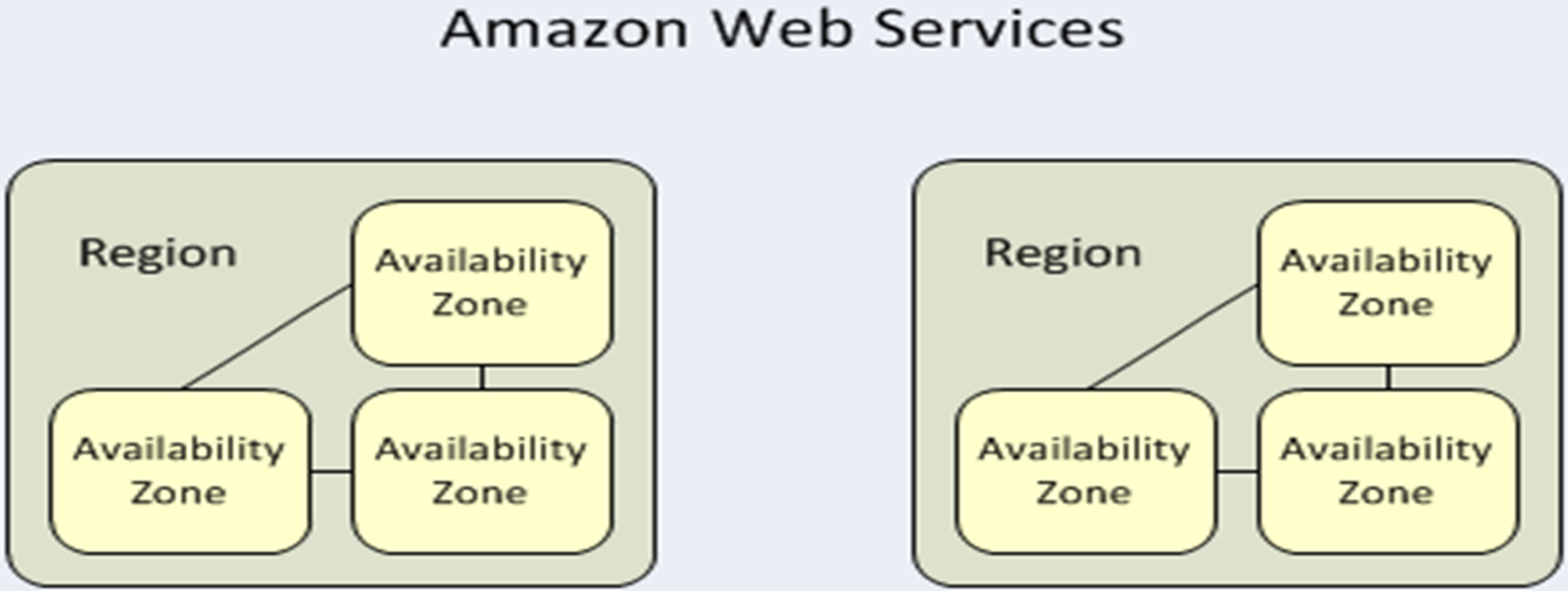
**Scalability:** Facing some resource issues, no problem within seconds you can scale up the resources and improve your application performance. This cannot be compared with traditional IT datacenters.

**Multiple OS's:** Choice and use any supported Operating systems.

**Multiple Storage Options:** Choice of high I/O storage, low cost storage. All is available in AWS, use and pay what you want to use with almost any scalability.

**Secure:** AWS is PCI DSS Level1, ISO 27001, FISMA Moderate, HIPAA, SAS 70 Type II passed. In-fact systems based on AWS are usually more secure than in-house IT infrastructure systems.

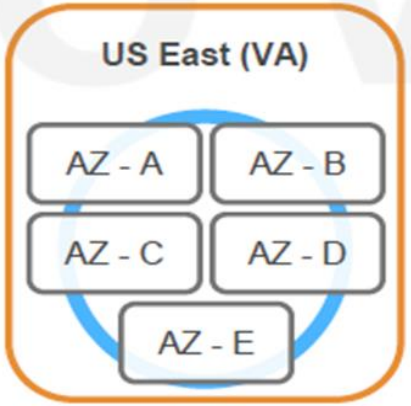
# Amazon Web Services

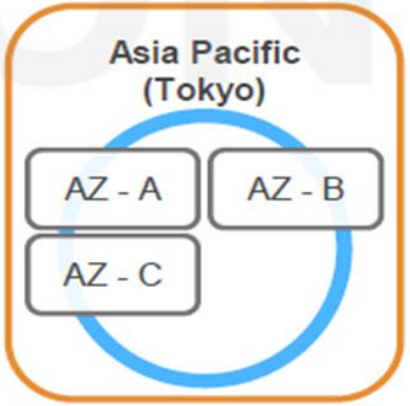


# Amazon Web Services

At least 2 AZs per region.

- Examples:
  - US East (N. Virginia)
    - us-east-1a
    - us-east-1b
    - us-east-1c
    - us-east-1d
    - us-east-1e
  - Asia Pacific (Tokyo)
    - ap-northeast-1a
    - ap-northeast-1b
    - ap-northeast-1c




*Note: Conceptual drawing only. The number of Availability Zones (AZ) may vary.*

# Amazon Web Services

**AWS Regions:**
-Geographic Locations
-Consists of at least two Availability Zones(AZs)
-All of the regions are completely independent of each other with separate Power Sources, Cooling and Internet connectivity.

**AWS Availability Zones**
-AZ is a distinct location within a region
-Each Availability Zone is isolated, but the Availability Zones in a Region are connected through low-latency links.
-Each Region has minimum two AZ’s
-Most of the services/resources are replicated across AZs for HA/DR purpose.

# Amazon Web Services

**AWS Regions:**

-Geographic Locations

-Consists of at least two Availability Zones(AZs)

-All of the regions are completely independent of each other with separate Power Sources, Cooling and Internet connectivity.

-This achieves the greatest possible fault tolerance and stability.

-There is a charge for data transfer between Regions.

-When you view your resources, you'll only see the resources tied to the Region you've specified.

-An AWS account provides multiple Regions so that you can launch Amazon EC2 instances in locations that meet your requirements. For example, you might want to launch instances in Europe to be closer to your European customers or to meet legal requirements.

-Resources aren't replicated across regions unless you do so specifically.

# Amazon Web Services

**AWS Availability Zones**

-AZ is a distinct location within a region

-Each Availability Zone is isolated, but the Availability Zones in a Region are connected through low-latency links.

-Each Region has minimum two AZ’s

-Most of the services/resources are replicated across AZs for HA/DR purpose.

-While launching instance you should specify an Availability Zone if your new instances must be close to, or separated from, your running instances.

# Amazon Web Services

## Current:

22 AWS Regions

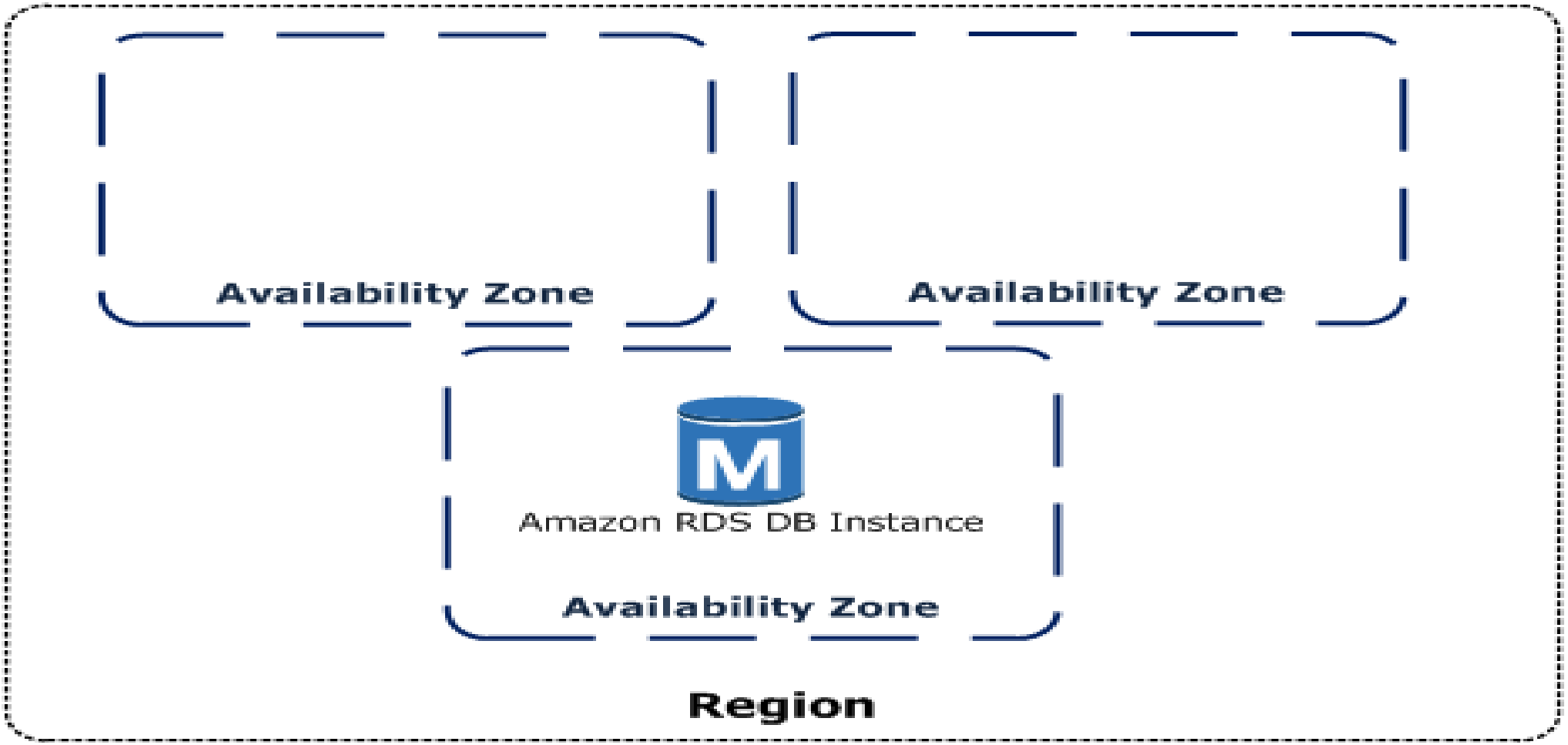
69 AZs

## Upcoming:

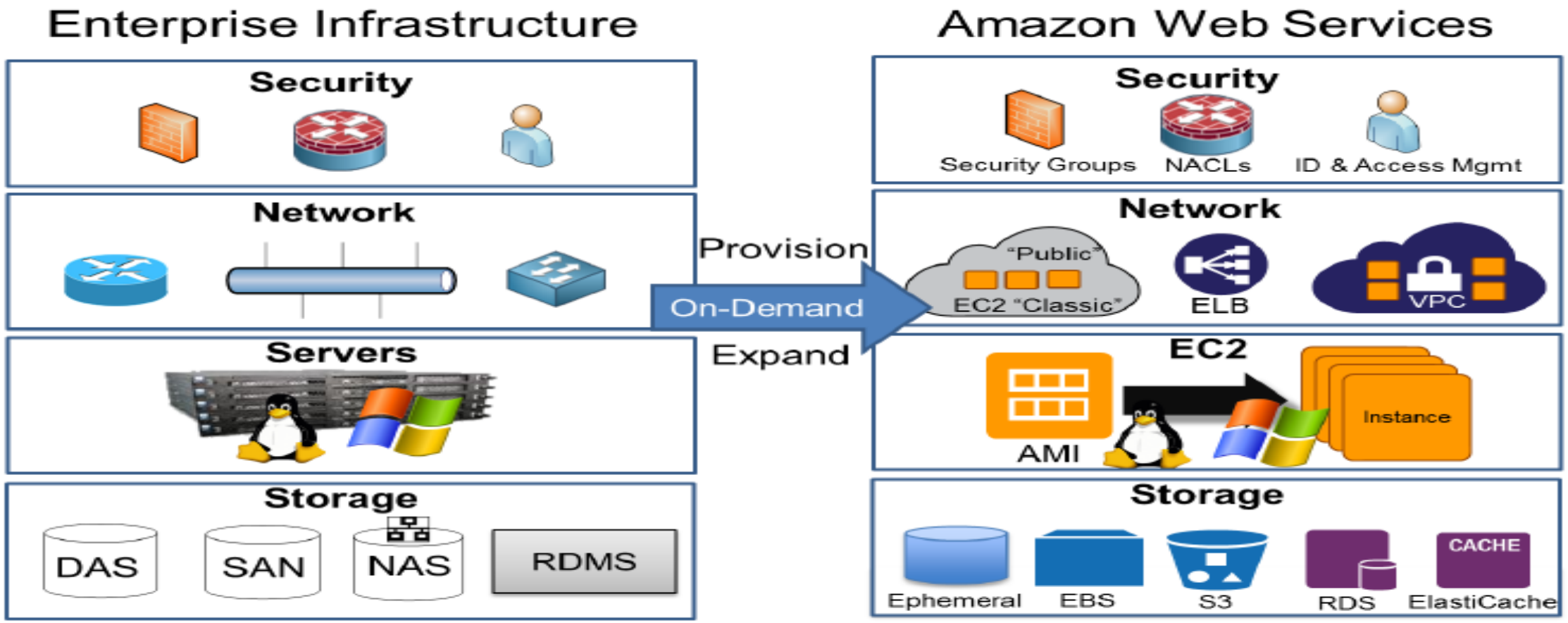
4 Regions

13 AZs

# Amazon Web Services



# AWS


Enterprise Infrastructure
Security
Network
Servers
Storage
DAS
SAN
NAS
RDMS
Provision
On-Demand
Expand
Amazon Web Services
Security
Security Groups
NACLs
ID & Access Mgmt
Network
"Public"
EC2 "Classic"
ELB
VPC
EC2
AMI
Instance
Storage
Ephemeral
EBS
S3
RDS
CACHE
ElastiCache

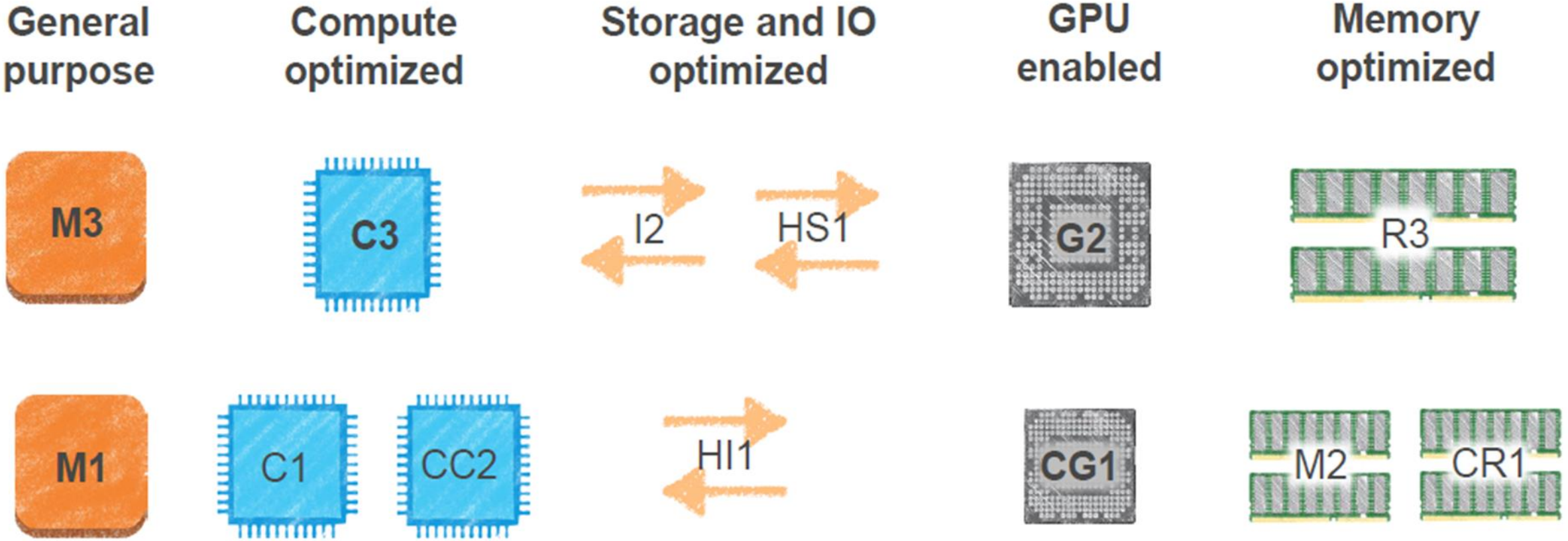
# AWS
# Compute Services

# AWS Elastic Compute Cloud

- Amazon EC2 stands for Elastic Compute Cloud, and is the Primary AWS web service.
- Provides Resizable compute capacity
- Reduces the time required to obtain and boot  new server instances to minutes
- There are two key concepts to Launch instances in AWS:
  - Instance Type
  - AMI

- EC2 Facts:
  - Scale capacity as your computing requirements change
  - Pay only for capacity that you actually use
  - Choose Linux or Windows OS as per need. You have to Manage the OS and Security of same.
  - Deploy across AWS Regions and Availability Zones for reliability/HA

# AWS EC2


General purpose
Compute optimized
Storage and IO optimized
GPU enabled
Memory optimized
M3
C3
I2
HS1
G2
R3
M1
C1
CC2
HI1
CG1
M2
CR1

# EC2 Security Group

Security Group is a Virtual Firewall Protection.

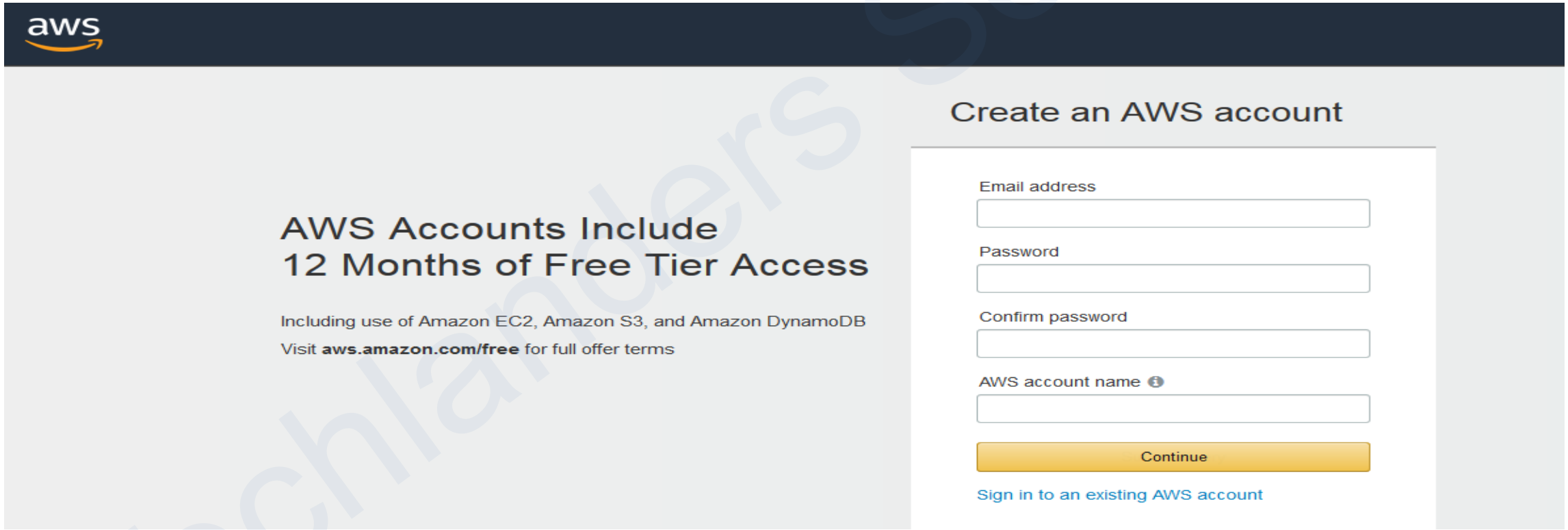
AWS allows you to control traffic in and out of your instances through virtual firewalls called security groups.

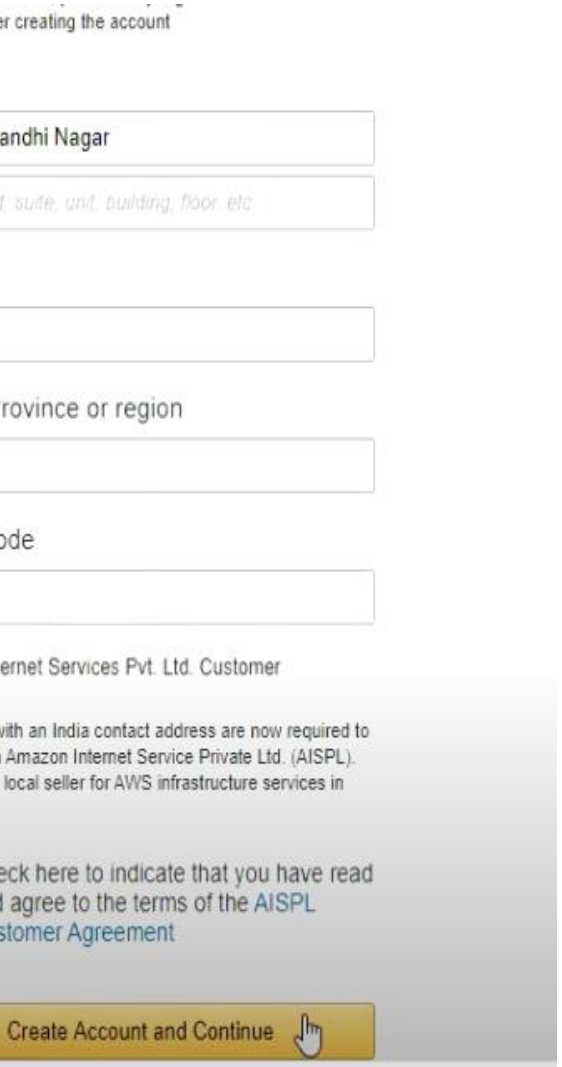
Security groups allow you to control traffic based on port, protocol, and source(inbound)/destination(outbound).

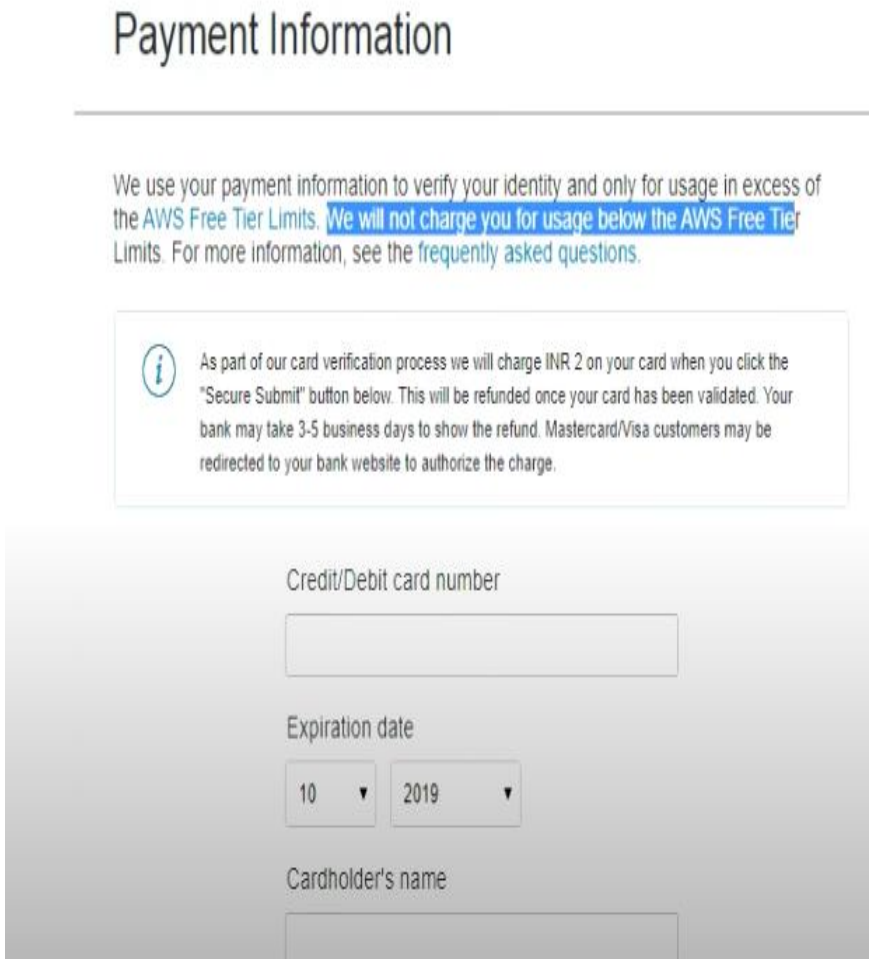
Security groups are associated with instances when they are launched. Every instance must have at least one security group. Though they can have more.

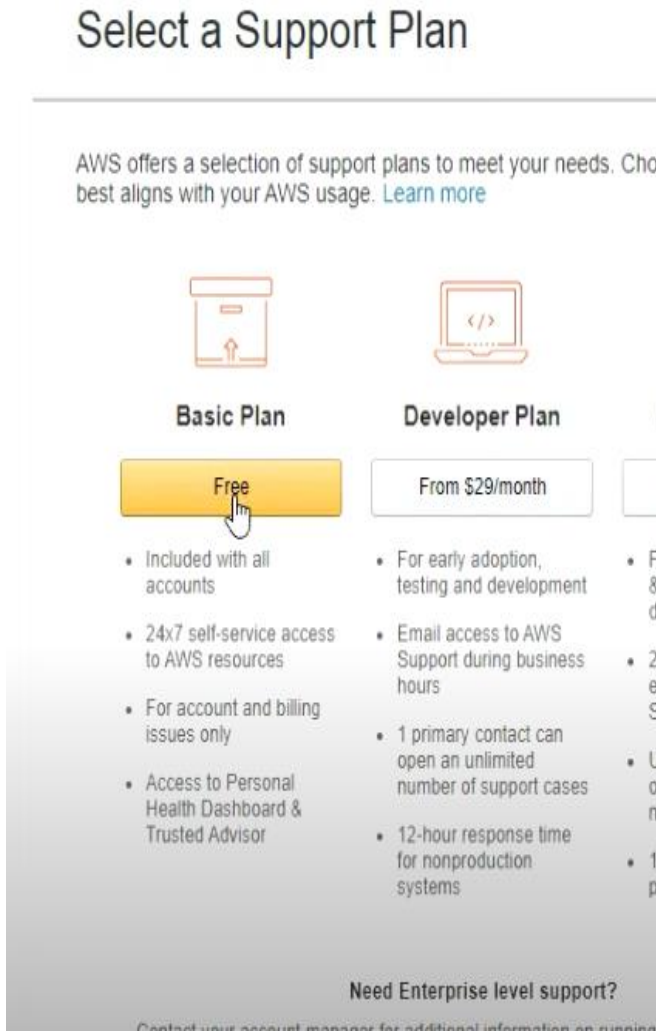
A security group is default deny.

# LAB 1





# Installation of Terraform on AWS Env.

# Terraform Fundamentals

# AWS CLI

# AWS CLI

AWS CLI is a command based utility to manage AWS resources

The primary distribution method for the AWS CLI on Linux, Windows, and macOS is pip, a package manager for Python that provides an easy way to install, upgrade, and remove Python packages and their dependencies

http://docs.aws.amazon.com/cli/latest/userguide/installing.html

Requirements

- Python 2 version 2.6.5+ or Python 3 version 3.3+
- Windows, Linux, macOS, or Unix
- Pip package should be present (else install python-pip)

Install AWSCLI: pip install awscli --upgrade --user
For Windows, directly download the Windows installer from CLI webpage

# AWS CLI

Lets install an AWSCLI
https://aws.amazon.com/cli

aws --version

aws help

aws ec2 help / aws s3 help / aws <anysubcommand> help

Configure your default keys and region:

root@ip-172-31-28-145:~# aws configure
AWS Access Key ID [None]: ##########
AWS Secret Access Key [None]: ################
Default region name [None]: us-west-2
Default output format [None]:
root@ip-172-31-28-145:~#

# LAB 2: AWS CLI

Check the details for all running instances using CLI

- aws ec2 describe-instances | grep -i instanceID

Creation of an AWS Instance using CLI:

- aws ec2 run-instances --image-id ami-05fa00d4c63e32376 --instance-type t2.micro --key-name raman
- aws ec2 stop-instances --instance-ids i-02fedc26aa77154a6
- aws ec2 terminate-instances --instance-ids i-02fedc26aa77154a6
- aws s3 ls
- aws iam list-users

# Providers

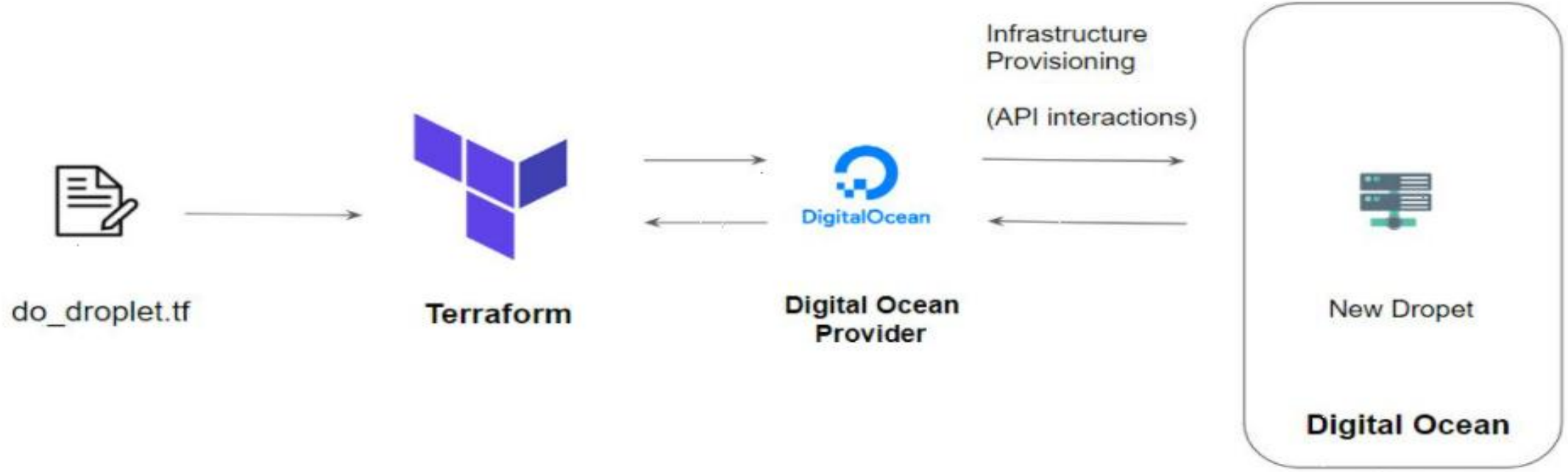
A provider is responsible for understanding API interactions and exposing resources over to a particular cloud service provider. Most providers configure a specific infrastructure platform (either cloud or self-hosted).

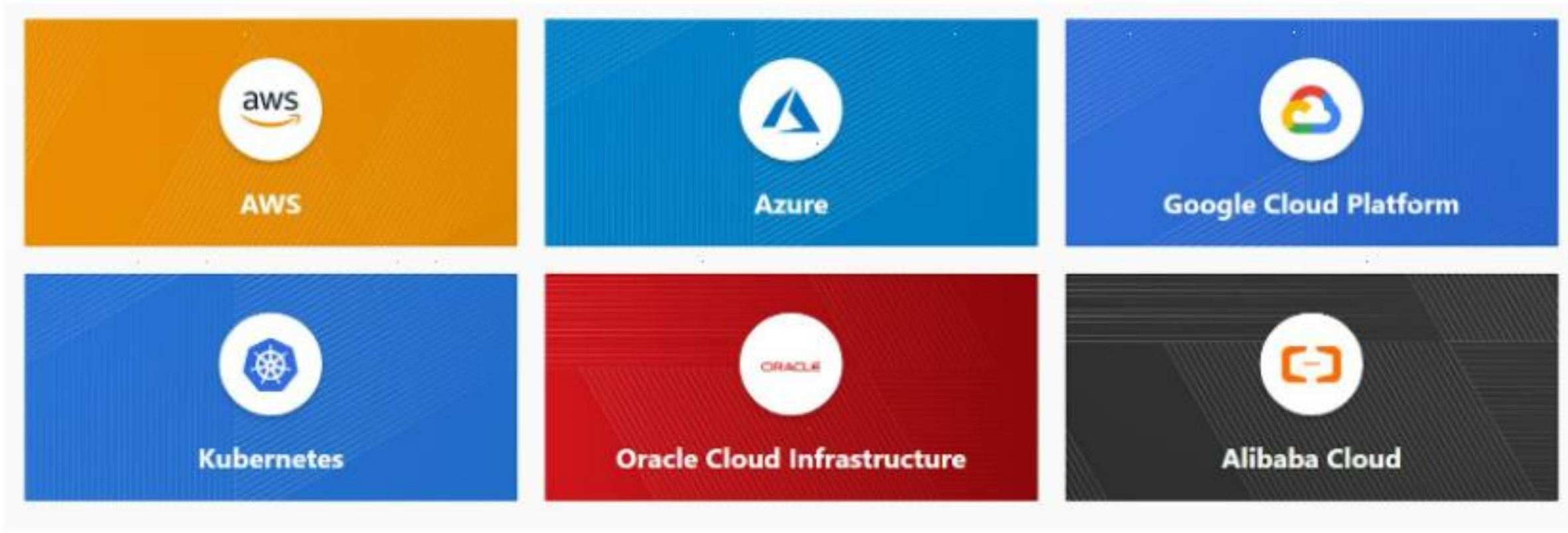
```
provider "aws" {
  region     = "us-east-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}
```

A provider is responsible for creating and managing resources.

https://registry.terraform.io/browse/providers

# Overview of Provider Architecture :




aws
AWS
Azure
Google Cloud Platform
Kubernetes
ORACLE
Oracle Cloud Infrastructure
Alibaba Cloud

# Resources

- Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, etc
- 
```
resource "aws_instance" "web" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
}
```

A resource block declares a resource of a given type ("aws_instance") with a given local name ("web"). The name is used to refer to this resource from elsewhere in the same Terraform module but has no significance outside that module's scope.

The resource type and name together serve as an identifier for a given resource and so must be unique within a module.

Resource names must start with a letter or underscore, and may contain only letters, digits, underscores, and dashes.

# LAB 3: Creating first ec2 instance

..

- https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance

# Configuration files

- Whatever you want to achieve(deploy) using terraform will be achieved with configuration files.
- Configuration files ends with .tf extension (tf.json for json version).
- Terraform uses its own configuration language, designed to allow concise descriptions of infrastructure.
- The Terraform language is declarative, describing an intended goal rather than the steps to reach that goal.
- A group of resources can be gathered into a module, which creates a larger unit of configuration.
- As Terraform's configuration language is declarative, the ordering of blocks is generally not significant. Terraform automatically processes resources in the correct order based on relationships defined between them in configuration

# Example

- You can write up the terraform code in hashicorp Language – HCL.
- Your configuration file will always endup with .tf extension

```
provider "aws" {
  region     = "us-east-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}

resource "aws_instance" "myec2" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
}
  tags = {
    Name = "Techlanders-aws-ec2-instance"
  }
}
```

# Terraform Workflow

**Few Steps to work with terraform:**

1) Set the Scope - Confirm what resources need to be created for a given project.
2) Author - Create the configuration file in HCL based on the scoped parameters
3) Run terraform init to initialize the plugins and modules
4) Run terraform validate to validate the template
5) Do terraform plan
6) Run terraform apply to apply the changes

# Terraform validate

- Terraform validate will validate the terraform configuration file

- It'll through error for syntax issues:

```
[root@TechLanders aws]# terraform validate
Success! The configuration is valid.

[root@TechLanders aws]#
```

# Terraform init

- Terraform init will initialize the modules and plugins.
- If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory.
- If you forget running init, terraform plan/apply will remind you about initialization.
- Terraform init will download the connection plugins from Repository “registry.terraform.io” under your current working directory/.terraform:

```
[root@TechLanders plugins]# pwd
/root/aws/.terraform/plugins
[root@TechLanders plugins]# ls -l
total 4
drwxr-xr-x. 3 root root  23 Aug 15 07:06 registry.terraform.io
-rw-r--r--. 1 root root 136 Aug 15 07:06 selections.json
[root@TechLanders plugins]#
```

- Important concept:
  - Always make a best practice to initialize the terraform modules with versions. i.e.
    hashicorp/aws: version = "~> 3.2.0"

# Example

- Perform Terraform Init:

```
[root@TechLanders aws]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v3.2.0...
- Installed hashicorp/aws v3.2.0 (signed by HashiCorp)

The following providers do not have any version constraints in configuration,
so the latest version was installed.
To prevent automatic upgrades to new major versions that may contain breaking
changes, we recommend adding version constraints in a required_providers block
in your configuration, with the constraint strings suggested below.
* hashicorp/aws: version = "~> 3.2.0"

Terraform has been successfully initialized!

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[root@TechLanders aws]#
```

# Terraform plan

- terraform plan will create an execution plan and will update you what changes it going to make.
- It'll update you upfront what its gonna add, change or destroy.
- Terraform will automatically resolve the dependency between components- which to be created first and which in last.

```
[root@TechLanders aws]# terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan but will not be persisted to local or remote state storage.
An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # aws_instance.myserver will be created
  + resource "aws_instance" "myserver" {
      + ami                          = "ami-06b35f67f1340a795"
      + arn                          = (known after apply)
Plan: 1 to add, 0 to change, 0 to destroy.
```

# Terraform apply

- Terraform apply will apply the changes.
- Before it applies changes, it'll showcase changes again and will ask to confirm to move ahead:

```
[root@TechLanders aws]# terraform apply
An execution plan has been generated and is shown below. Resource actions are indicated with the following symbols:
  + create
Do you want to perform these actions?   Terraform will perform the actions described above.   Only 'yes' will be accepted to approve.
  Enter a value: yes
aws_instance.myserver: Creating...
aws_instance.myserver: Still creating... [10s elapsed]
aws_instance.myserver: Still creating... [20s elapsed]
aws_instance.myserver: Creation complete after 21s [id=i-0a63756c96d338801]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@TechLanders aws]#
```

# Terraform apply

- Terraform apply will create **tfstate** file to maintain the desired state:

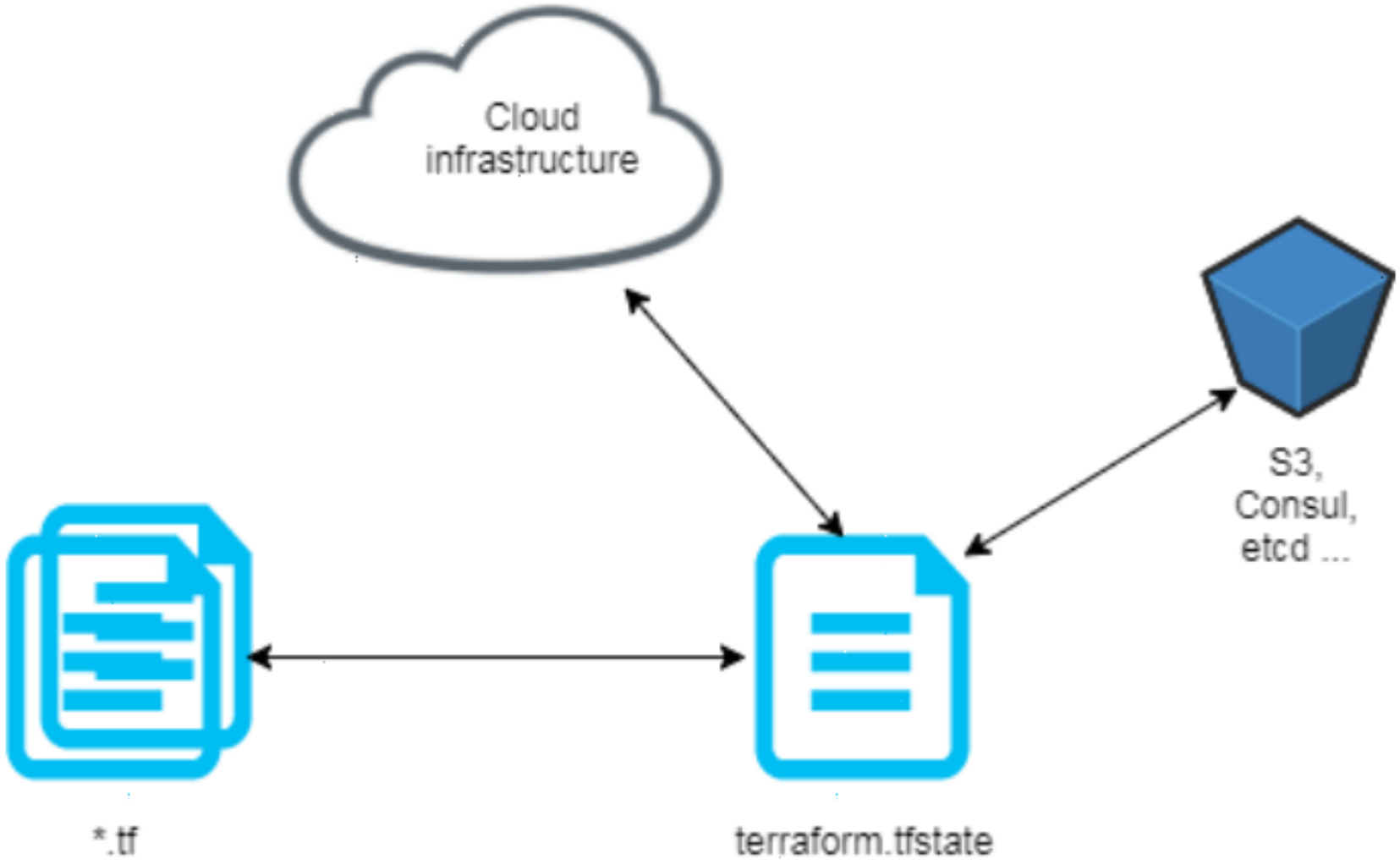
```
[root@TechLanders aws]# ls -l
total 8
-rw-r--r--. 1 root root  234 Aug 15 07:06 myinfra.tf
-rw-r--r--. 1 root root 3209 Aug 15 08:02 terraform.tfstate
[root@TechLanders aws]# cat terraform.tfstate
{
 "version": 4,
 "terraform_version": "0.13.0",
 "serial": 1,
 "lineage": "7f7e0e15-95ef-d8fa-b1cd-12024aed5fa6",
 "outputs": {},
 "resources": [
  "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
    "instances": [
     {
       "schema_version": 1,
       "attributes": {
         "ami": "ami-06b35f67f1340a795",
         "arn": "arn:aws:ec2:us-east-2:677729060277:instance/i-0a63756c96d338801",
```

- Note: -auto-approve option can be given alongwith terraform apply to avoid the human intervention.

# Terraform show

- Terraform show will show the current state of the environment been created by your config file:

```
[root@ip-172-31-6-233 aws]# terraform show
# aws_instance.myserver:
resource "aws_instance" "myserver" {
    ami                          = "ami-06b35f67f1340a795"
    arn                          = "arn:aws:ec2:us-east-2:677729060277:instance/i-0a63756c96d338801"
    associate_public_ip_address  = true
    availability_zone            = "us-east-2a"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
----
----
```


Cloud
infrastructure
S3,
Consul,
etcd ...
*.tf
terraform.tfstate

# Desired State Maintenance (DSC)

- Delete the newly created server and then check for the terraform plan

```
[root@TechLanders aws]# terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # aws_instance.myserver will be created
  + resource "aws_instance" "myserver" {
```

- Run terraform apply command again and witness the provisioning of new server on console.

```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create
Terraform will perform the following actions:
  # aws_instance.myserver will be created
```

# Infrastructure as Code

- Modify your template file to change the instance size from t2.micro to t2.small and plan/apply the changes:

```
[root@TechLanders aws]# cat myinfra.tf
resource "aws_instance" "myserver" {
ami = "ami-06b35f67f1340a795"
instance_type = "t2.small"
}
[root@TechLanders aws]#
```

- Run terraform plan and apply again to check the differences

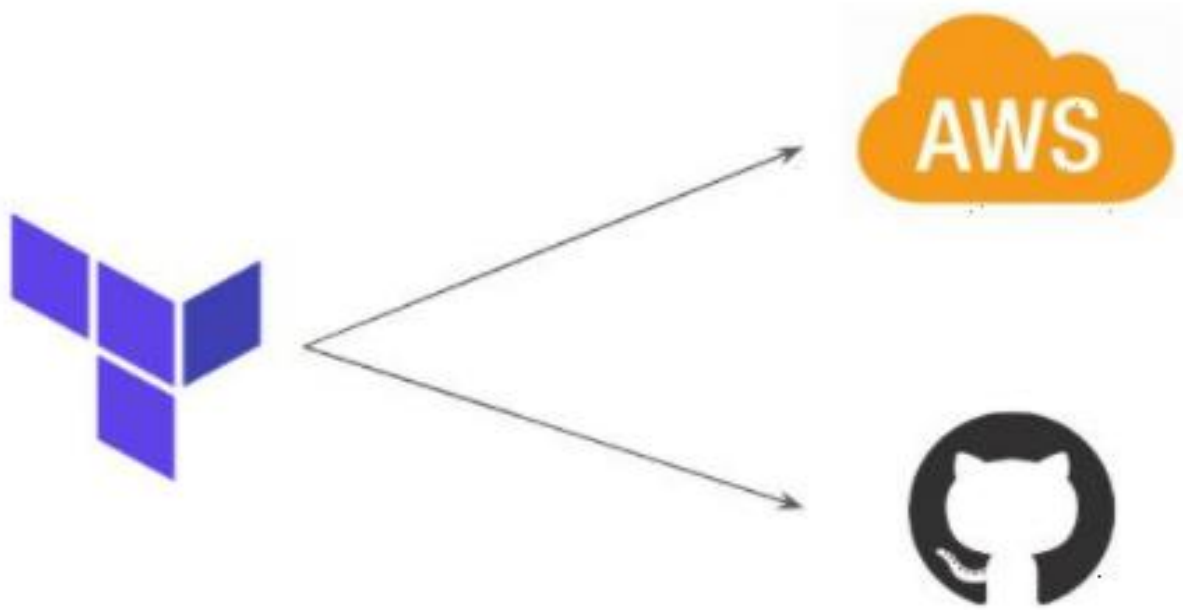
```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a1f8a600cb968c7c]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  ~ update in-place
Plan: 0 to add, 1 to change, 0 to destroy.
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.
  Enter a value: yes
aws_instance.myserver: Modifying... [id=i-0a1f8a600cb968c7c]
```

# Refreshing the state

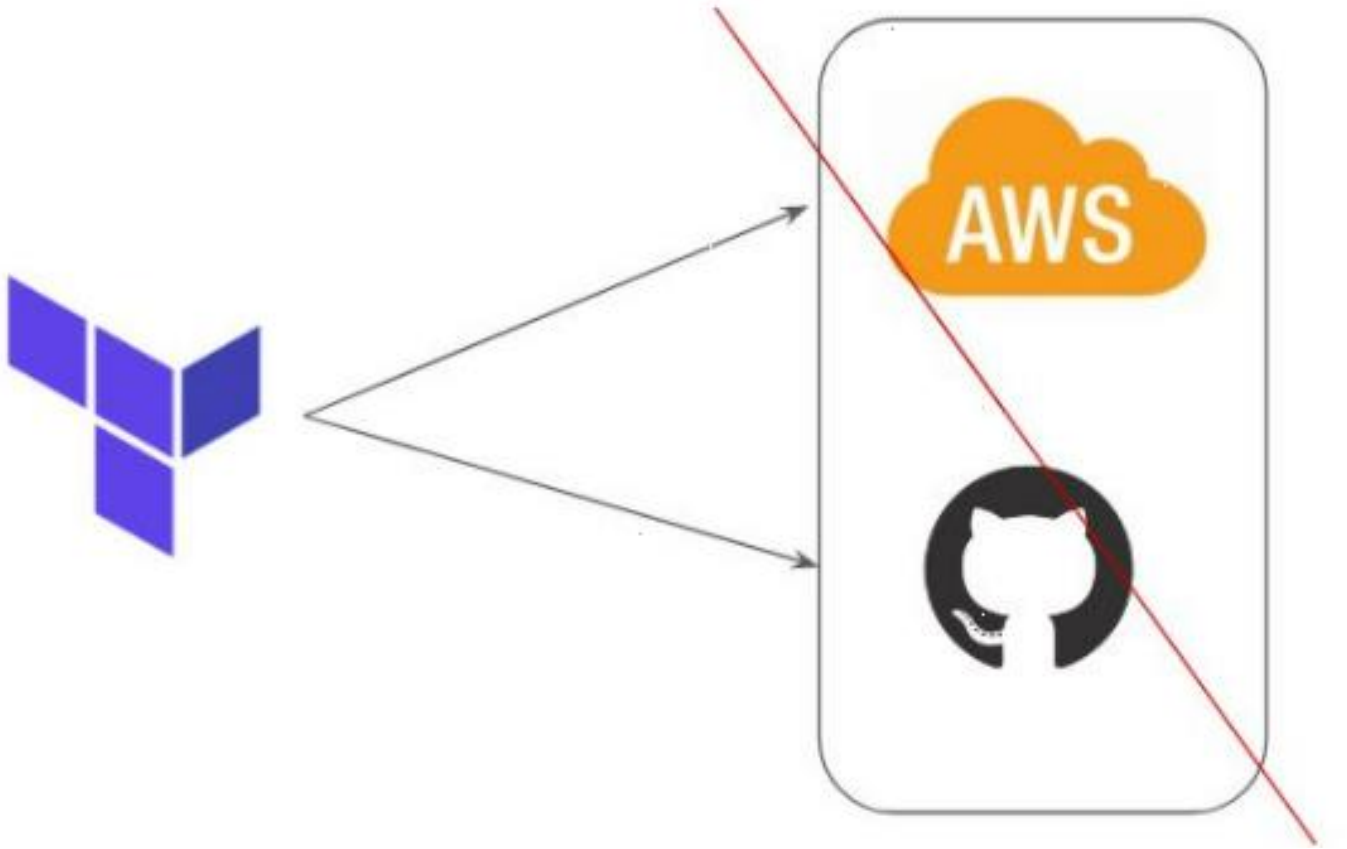
- In case the requirement is to just check for any updates been done in the running environment, we can run terraform refresh command:

```
C:\Users\gagandeep\Desktop\terraform>terraform refresh

google_compute_network.vpc_network: Refreshing state... [id=projects/accenture-286519/global/networks/terraform-net3]

google_compute_address.vm_static_ip: Refreshing state... [id=projects/accenture-286519/regions/us-central1/addresses/terraform-static-ip1]

google_compute_instance.vm_instance1: Refreshing state... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance1]


C:\Users\gagandeep\Desktop\terraform>
```

# Lab 4: Working with other providers ..



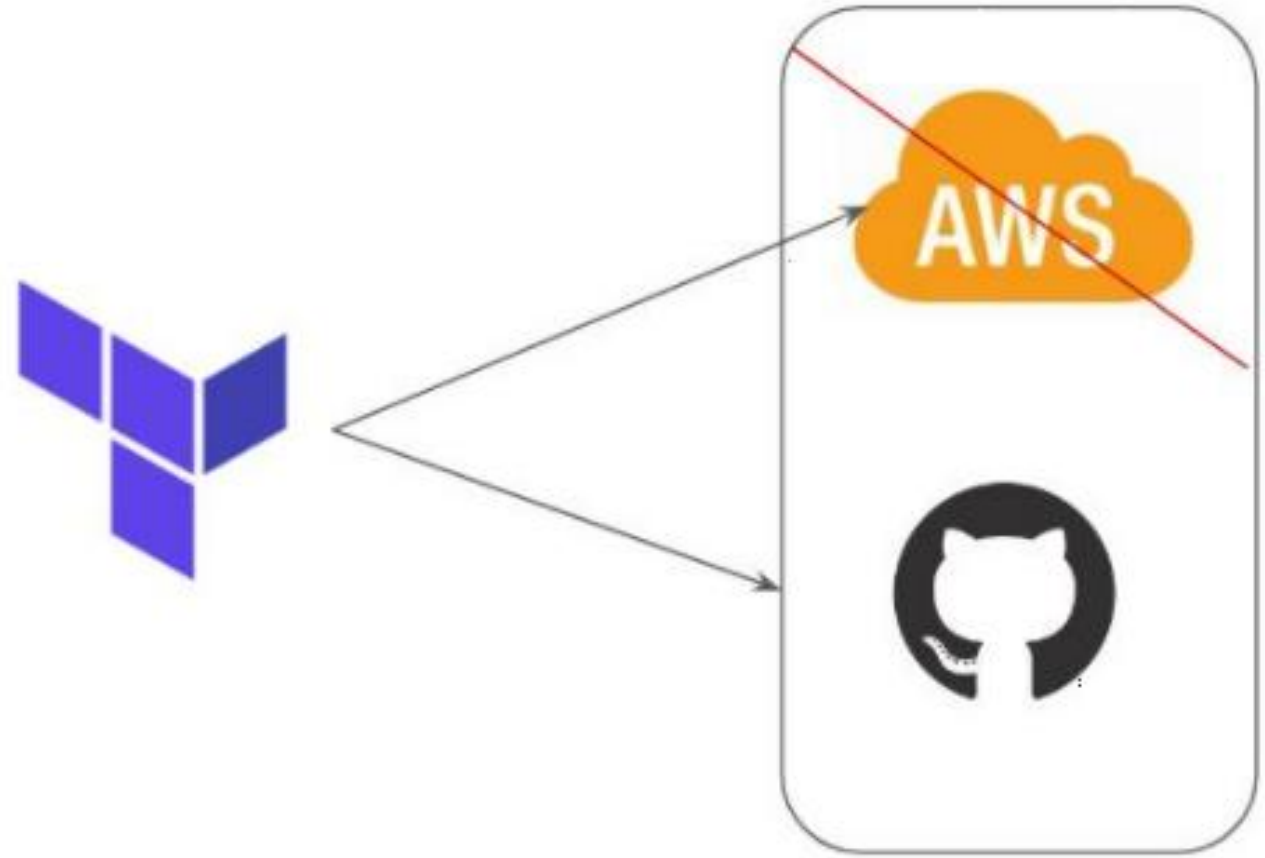
# Destroying Infra in one go :



# Destroying Infra in one go

- Terraform destroy will destroy the infrastructure in one go by using your tfstate file.

```
[root@TechLanders aws]# terraform destroy
aws_instance.myserver: Refreshing state... [id=i-0a1f8a600cb968c7c]
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  - destroy
Terraform will perform the following actions:
  # aws_instance.myserver will be destroyed
  - resource "aws_instance" "myserver" {
      - ami  = "ami-06b35f67f1340a795“
 Enter a value: yes
aws_instance.myserver: Destroying... [id=i-0a1f8a600cb968c7c]
aws_instance.myserver: Still destroying... [id=i-0a1f8a600cb968c7c, 10s elapsed]
aws_instance.myserver: Still destroying... [id=i-0a1f8a600cb968c7c, 20s elapsed]
aws_instance.myserver: Destruction complete after 29s
Destroy complete! Resources: 1 destroyed.
```


AWS

# Destroying Infra

- Terraform destroy can also delete selected resources given with –target option and can also be auto-approved with –auto-approve option. But it is always recommended to modify the configuration file instead of –target.

terraform destroy -target github_repository.repo

github_repository.repo: Refreshing state... [id=terraform-repo]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

# github_repository.repo will be destroyed

- resource "github_repository" "repo" {

- allow_auto_merge = false -> null

, which means that the result of this plan may not represent all of the changes requested by the current configuration.

The -target option is not for routine use and is provided only for exceptional situations such as recovering from errors or mistakes, or when Terraform specifically suggests to use it as part of an error message.

Note: Multiple –target options are supported as well.

# Lab 5 : Desired ,current state and last known configuration ..

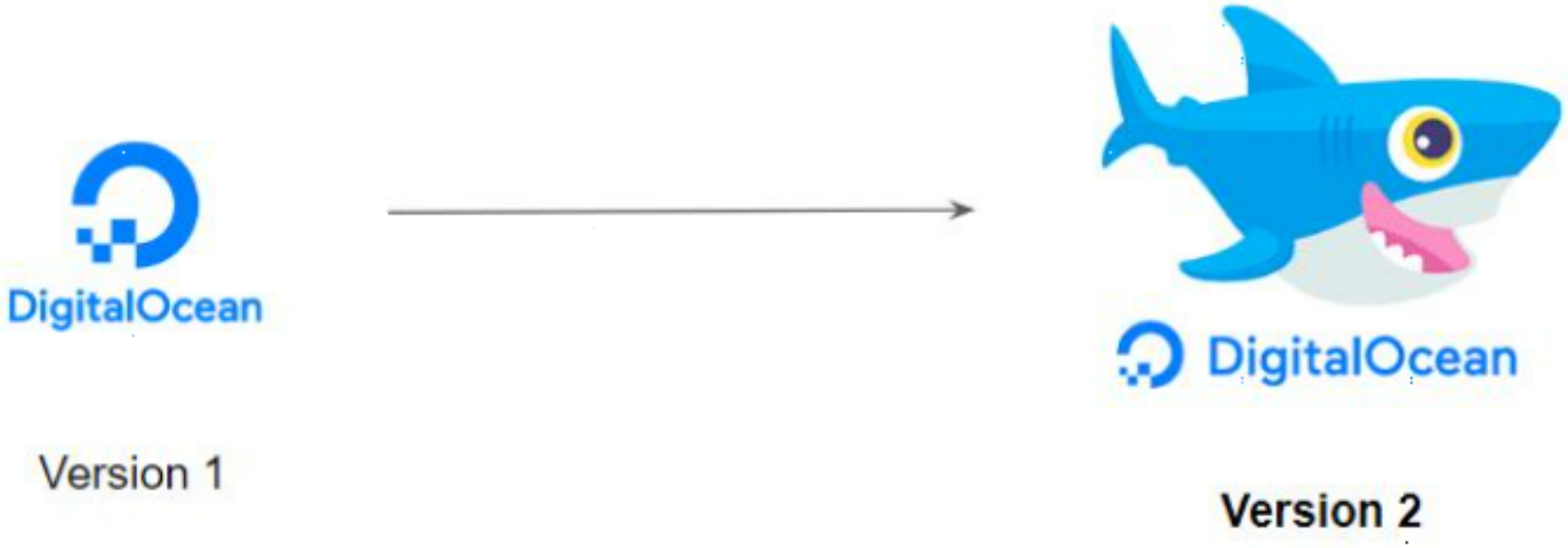
Terraform tries to ensure that the deployed infrastructure is based on the desired state.

If there is a difference between the two, terraform plan presents a description of the changes necessary to achieve the desired state.

# LAB 6:CHALLENGE WITH DESIRED AND CURRENT STATE ..

Provider plugins are released separately from Terraform itself.

They have a different set of version numbers.



# PROVIDER VERSIONING :

- Different Version Parameters :
- version = "2.7"
- version = ">= 2.8“
- version = “~> 2.x”
- version = "<= 2.8"
- version = ">=2.10,<=2.30"

# LAB 7: PROVIDER VERSIONING ..

# Output from a run

Terraform provides output for every run and same can be used to list the resources details which are created using help of Terraform:

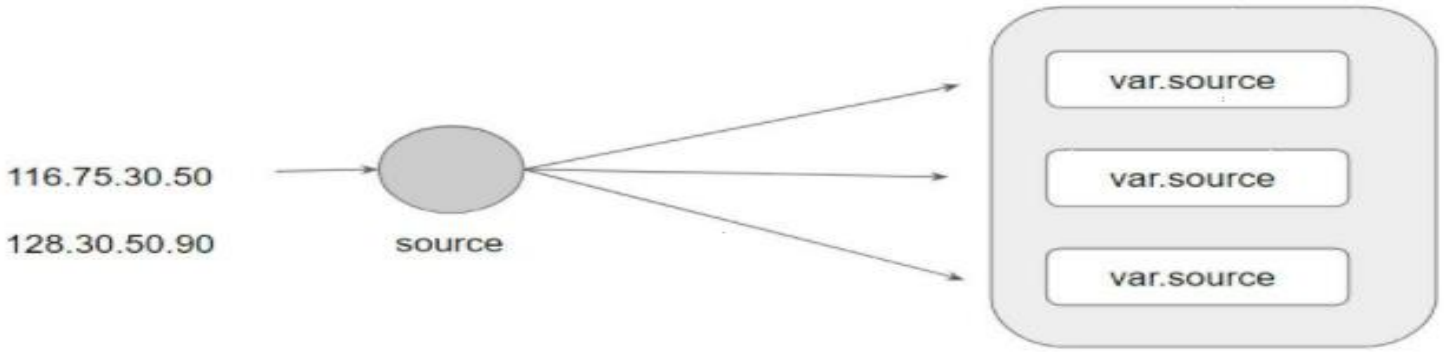
```
output "myawsserver-ip" {
  value = [aws_instance.myawsserver.public_ip]
}
```

# LAB 8 : EC2 instance with output value ..

# Variables

Repeated static values can create more work in the future.


Terraform Variables allows us to centrally define the values that can be used in multiple terraform configuration blocks.



# Lab 9: Variables ..

# Variables

- Variables can be of different types, based on terraform versions:
  - Strings

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

  - Numbers

```
variable "web_instance_count" {
 type    = number
 default = 1 }
```

  - Lists

```
variable "cidrs" { default = ["10.0.0.0/16”] }
```

  - Maps

```
variable "machine_types" {
 type    = map
 default = {
  dev  = "f1-micro"
  test = "n1-highcpu-32"
  prod = "n1-highcpu-32"
 }
```

# *LAB 10: Different Approaches for Variable Assignment ..*

# Variables

- Variables can be assigned via different ways:
  - Via UI (if no default value is set in variable.tf)

    *Via variable.tf default value

    *Via .tfvars file (terraform.tfvars or custom.tfvars)
  - Via command line flags:

    terraform plan -var="instancetype=t2.small"

    terraform plan -var-file="custom.tfvars“

# Variables Definition Precedence

Terraform loads variables in the following order, with later sources taking precedence over earlier ones:

- Environment variables
- The terraform.tfvars file, if present.
- The terraform.tfvars.json file, if present.
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- Any -var and -var-file options on the command line, in the order they are provided. (This includes variables set by a Terraform Cloud workspace.)

# Lab 11: Variables

1) Declare AMI as variable and use same in your aws_instance resource

2) Define the value of AMI( with AMIID) inside terraform.tfvars file

3) terraform plan

4) Rename terraform.tfvars with abc.tfvars

5) Run terraform plan again

6) Run terraform plan with -var-file=abc.tfvars and see the outputs

7) Run terraform plan with -var ami="AMI_ID"

# Executions

```
[root@main-tf app1]# terraform plan -var-file="abc.tfvars"

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated
with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myawsserver will be created
  + resource "aws_instance" "myawsserver" {
      + ami                          = "ami-064ff912f78e3e561"
```

# LAB 12: Fetching data from map and list in variable ..

# Working with change

Changes are of two types:

- Up-date In-place
- Disruptive

So always be careful with what you are adding/modifying

# Update in-place

Update in-place will ensure your existing resources intact and modify the existing resources only. Here also based on what configuration is required to be changed, server may or may-not shutdown.

Making an update in your infra so that your resource state does not get affected.

(1 change)

# LAB 13 :UPDATE IN PLACE ..

# Update - Disruptive

Disruptive updates require a resource to be deleted and recreated.

For example, modifying the image type for an instance will require instance to be deleted and re-created.

Making an update in your infra so that your resource state does not get affected.

(1 change)

update Disruptive:

Making an update in your infra so that old resource gets terminated/destroyed and a new resource gets created/deployed

( 1 add, 1 destroy)

# LAB 14 :UPDATE DISRUPTIVE ..

# LAB 15 : Changes outside of terraform

Changes which occurred outside of terraform are unwanted changes and if anything which is modified outside of terraform is detected, same will be marked in state files and will be corrected at next apply.

- Create a terraform code creating a server (code on next page)
- Run terraform show command to check current required state of infrastructure.
- Modify the tag through management console .
- Run terraform plan to check the behavior of terraform against the changes
- Check the terraform show command to view state file
- Check terraform refresh command to update the state frontend
- Run terraform apply to revert the changes
- Check the terraform refresh/show command as well as console again to validate the reversion of changes.

```
[root@main-tf app1]# cat fetch.tf
provider "aws" {
  region     = "us-east-2"
}

resource "aws_instance" "myec2" {
   ami = "ami-064ff912f78e3e561"
   instance_type = "t2.micro"
tags = {
Name = "raman-server"
}
}
```

# COUNT and INDEXING :

Count is a meta-argument defined by the Terraform language. It can be used with modules and with every resource type. The count meta-argument accepts a whole number, and creates that many instances of the resource or module.

# LAB 16 : COUNT

..

# LOCAL VALUES :

Terraform locals are named values that you can refer to in your configuration. You can use local values to simplify your Terraform configuration and avoid repetition. Local values (locals) can also help you write more readable configuration by using meaningful names rather than hard-coding values.

# LAB 17: LOCAL VALUES

# Terraform Functions

# Functions

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values.

Functions help us to perform few specific tasks (i.e. sort, search, reads, dates etc) easily with pre-written programs.

The Terraform language has a number of built-in functions that can be used in expressions to transform and combine values. Functions follow a common syntax:

<FUNCTION NAME>(<ARGUMENT 1>, <ARGUMENT 2>)

For e.g.

```
min(55, 3453, 2)
```

# LAB 18 : Functions

..

# LAB 19 : DATA SOURCES ..

# DEBUGGING:

Terraform has detailed logs which can be enabled by setting the TF_LOG environment variable to any value. This will cause detailed logs to appear on stderr.
You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR t o change the verbosity of the logs.

# LAB 20: DEBUGGING

..

# Terraform format

- The terraform fmt command is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.

# Dynamic Blocks

Terraform dynamic blocks are used to create repeatable nested blocks inside an argument. These dynamic blocks represent separate objects that are related or embedded with the containing object. Dynamic blocks are a lot like the for expression except dynamic blocks iterate over complex values.

# DYNAMIC BLOCK
# LAB 21 ..

# Node Tainting

# Tainting a Node

- In case there is a requirement to delete and recreate a resource, you can mark same in Terraform to tell terraform to do so. Terraform taint does so. We can manually mark a resource as tainted, forcing a destroy and recreate on the next plan/apply.
- Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource. For example: re-running provisioners will cause the node to be different or rebooting the machine from a base image will cause new startup scripts to run.
- Tainting a resource for recreation may affect resources that depend on the newly tainted resource

# Tainting a Node

# LAB 22: TAINT NODE

# Resource Dependencies

- There are two types of dependencies available in terraform:
  - Implicit - Dependency automatically detected and Hierarchy map automatically created by terraform
  - Explicit - The depends_on argument can be added to any resource and accepts a list of resources to create explicit dependencies on resources.
- Terraform uses dependency information to determine the correct order in which to create and update different resources.

# Implicit Dependencies

- Real-world infrastructure has a diverse set of resources and resource types.
- Dependencies among resources are obvious and should be maintained during provisioning. For e.g. Creating a network first than a Virtual machine; and creating a static IP before a VM is initialized and attaching that IP to it.

You can put your resources here and there in configuration file and terraform will automatically build a dependency map between them.

# Explicit Dependencies

- Sometimes there are dependencies between resources that are not visible to Terraform. The depends_on argument can be added to any resource and accepts a list of resources to create explicit dependencies for.
- For example, perhaps an application we will run on our instance expects to use a specific Cloud Storage bucket, but that dependency is configured inside the application code and thus not visible to Terraform. In that case, we can use depends_on to explicitly declare the dependency.

# LAB 23: DEPENDENCY ..

# Backup

- Just run terraform destroy or terraform apply and cancel it. Cross-check for terraform.tfstate.backup file which is being created as backup for your statefile.

```
C:\Users\gagandeep\terra>dir

16-08-2020  00:24    <DIR>          .
16-08-2020  00:24    <DIR>          ..
16-08-2020  00:12    <DIR>          .terraform
16-08-2020  00:24               226 .terraform.tfstate.lock.info
16-08-2020  00:08               243 myinfra.tf
15-08-2020  11:45        85,426,504 terraform.exe
16-08-2020  00:22             3,203 terraform.tfstate
16-08-2020  00:22             3,205 terraform.tfstate.backup
               5 File(s)     85,433,381 bytes
               3 Dir(s)  735,488,614,400 bytes free

C:\Users\gagandeep\terra>
```

# Terraform plan – saving plans

- Even you can save the terraform plan output for a later reference and then apply same to terraform apply command:

  C:\Users\gagandeep\terra>terraform plan -out t1
  Refreshing Terraform state in-memory prior to plan...
  The refreshed state will be used to calculate this plan, but will not be
  persisted to local or remote state storage.
  ---
  C:\Users\gagandeep\terra>terraform apply t1
  google_compute_address.vm_static_ip: Creating...

- You can create multiple plans and then execute one out of them, once you have finalized the stuff.
- After running terraform apply, your plan files become stale and can no longer be used.

  C:\Users\gagandeep\terra>terraform apply t1
  Error: Saved plan is stale

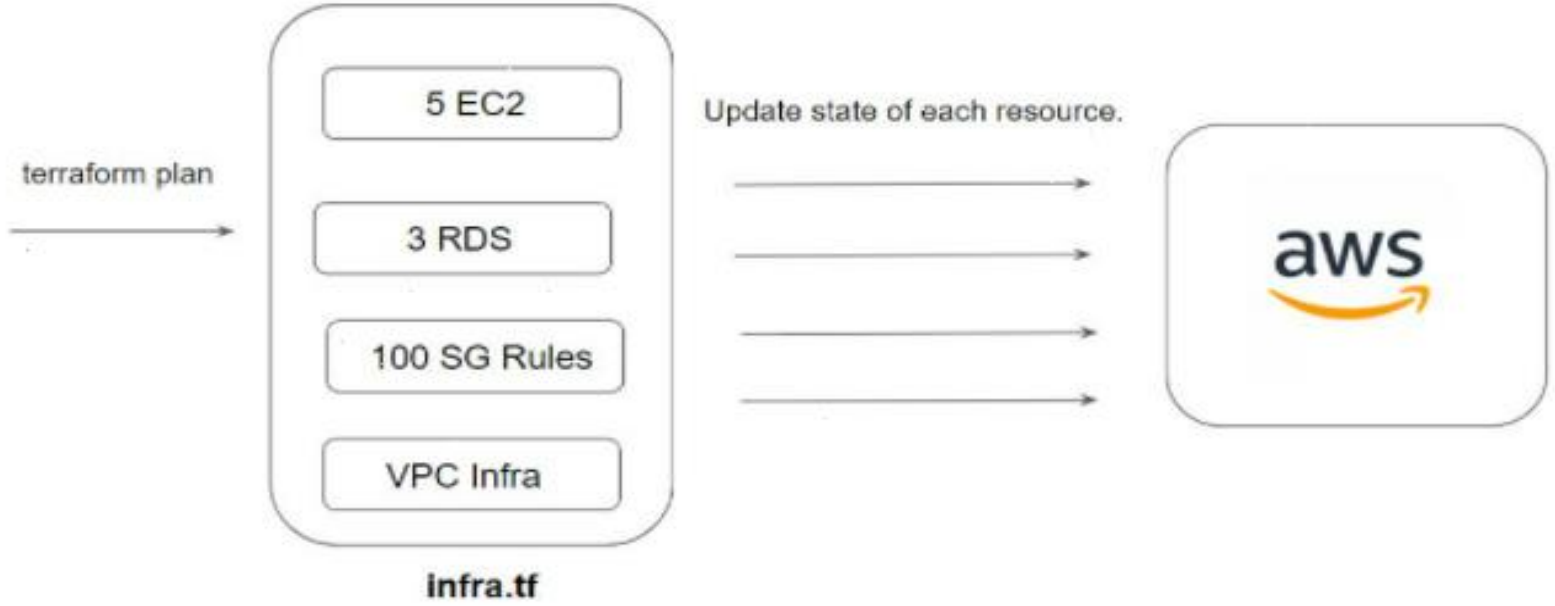
# Splat expressions :

- Splat expression [*] allows us to get a list of all the attributes of a resource .

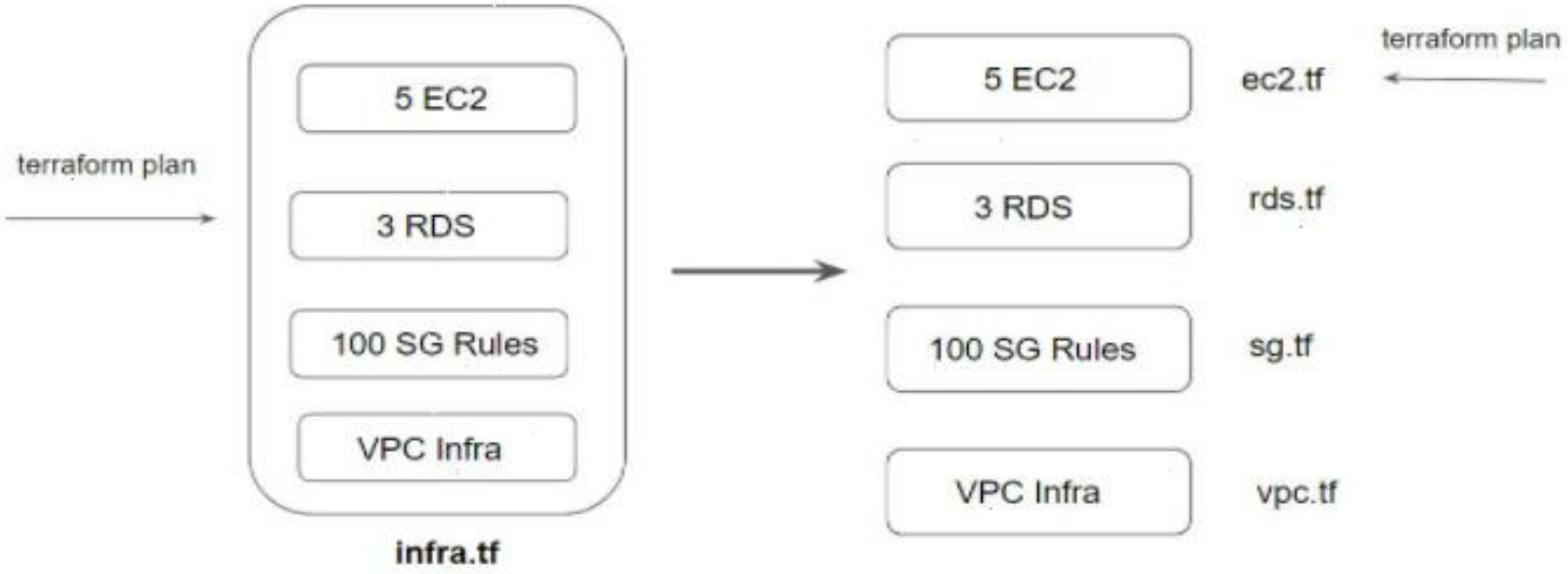
# LAB 24 : Splat expression ..

# DEALING WITH LARGE INFRA :

When you have a larger infrastructure, you will face issues related to API limits for a provider.

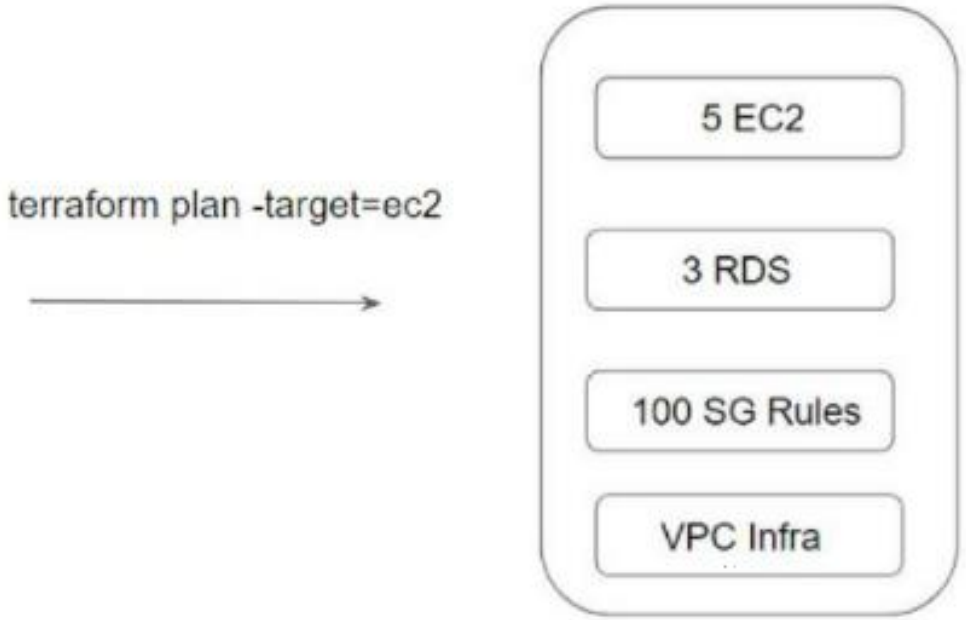


It is important to switch to a smaller configurations were each can be applied independently.



The -target=resource flag can be used to target a specific resource.

Generally used as a means to operate on isolated portions of very large configurations



# LAB 25 : LARGE INFRA ..

# Provisioners

- Terraform uses provisioners to upload files, run shell scripts, or install and trigger other software like configuration management tools.
- Multiple provisioner blocks can be added to define multiple provisioning steps.
- Terraform treats provisioners differently from other arguments. Provisioners only run when a resource is created but adding a provisioner does not force that resource to be destroyed and recreated.

# Provisioners

- Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.
- Running Provisioners can help you to execute stuff as per requirement
- The local-exec provisioner executes a command locally on the machine running Terraform, not the VM instance itself.
- Terraform don't encourage the use of provisioners, as they add complexity and uncertainty to terraform usage. Hashicorp recommends resolving your requirement using other techniques first, and use provisioners only if there is no other option left.
- When deploying virtual machines or other similar compute resources, we often need to pass in data about other related infrastructure that the software on that server will need to do its job.
- **Note:** Provisioners should only be used as a last resort. For most common situations there are better alternatives.

# Provisioners

- Provisioners also add a considerable amount of complexity and uncertainty to Terraform usage.
- Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action.
- Secondly, successful use of provisioners requires coordinating many more details than Terraform usage usually requires - direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.
- Some use cases:
  - Passing data into virtual machines and other compute resources
  - Running configuration management software

# Remote-Exec Provisioners

- Remote-Exec provisioner helps you to execute commands on remote machine:

# LAB 26: REMOTE EXEC

# Local-exec Provisioners

- Running Provisioners can help you to execute stuff as per requirement
- The local-exec provisioner executes a command locally on the machine running Terraform, not the VM instance itself.

# LAB 27 : LOCAL–EXEC ..

# FAILURE BEHAVIOR :

By default, provisioners that fail will also cause the terraform apply itself to fail.

The on_failure setting can be used to change this. The allowed values are:

| Allowed Values | Description |
|---|---|
| continue | Ignore the error and continue with creation or destruction. |
| fail | Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource. |

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command    = "echo The server's IP address is ${self.private_ip}"
    on_failure = continue
  }
}
```

# Alias :Multiple Providers

- Same Providers with multiple alias can be given for region or attributes change:

```
provider "aws" {
  region = "us-east-2"
  access_key = "AKIAJB2KQBDL56XQEYA"
  secret_key = "rNNWuzvBpp+v//XCB10Zr2OVuPI3iayxXXStPs"
  alias  = "useast2"
}

provider "aws" {
  region = "us-east-1"
  access_key = "AKIAJB2KQBD56XQEYA"
  secret_key = "rNNWuzvBpp//B10Zr2OVuPI3iayxXXStPs"
  alias  = "useast1"
}
```

# Multiple Providers

- Provide the provider name in resource:

```
resource "aws_instance" "myawsserver1" {
  ami = "ami-0c94855ba95c71c99"
  instance_type = "t2.micro"
  provider = aws.useast1
  tags = {
    Name = "Techlanders-aws-ec2-instance1"
    Env = "Prod"
  }
}
resource "aws_instance" "myawsserver2" {
  ami = "ami-0603cbe34fd08cb81"
  provider = aws.useast2
  instance_type = "t2.micro"

  tags = {
    Name = "Techlanders-aws-ec2-instance2"
    Env = "Prod"
  }
}
```

# LAB 28 : MULTIPLE PROVIDERS ..

- Create two servers  in two different regions using the same code using alias.

# Terraform Modules

# Terraform Modules

A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.

The .tf files in your working directory when you run terraform plan or terraform apply together form the root module. That module may call other modules and connect them together by passing output values from one to input values of another.

**Usual Structure:**

$ tree minimal-module/

.

├── README.md

├── main.tf

├── variables.tf

├── outputs.tf

We can centralize the terraform resources and can call out from TF files whenever required.


source

```
resource "aws_instance" "myweb" {
    ami = "ami-bf5540df"
   instance_type = "t2.micro"
   security_groups = ["default"]
}
```