Terraform Advanced with AWS/Azure

Terraform

Introduction

Your Name

Total experience

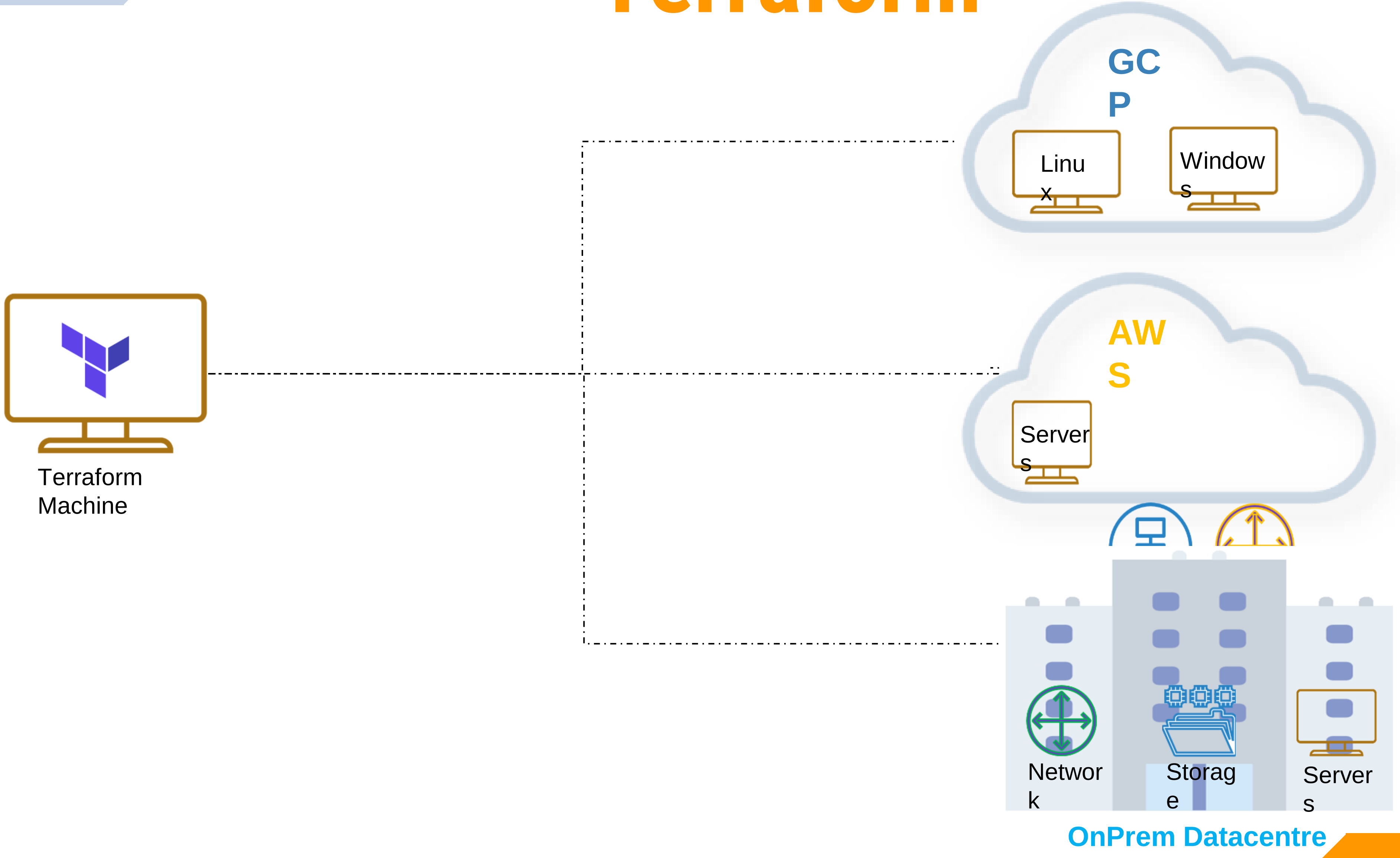
Background – Development / Infrastructure / Database / Network

Experience on Cloud, Terraform, Git, Jenkins

Your expectations from this training

What is Orchestration?

Terraform


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

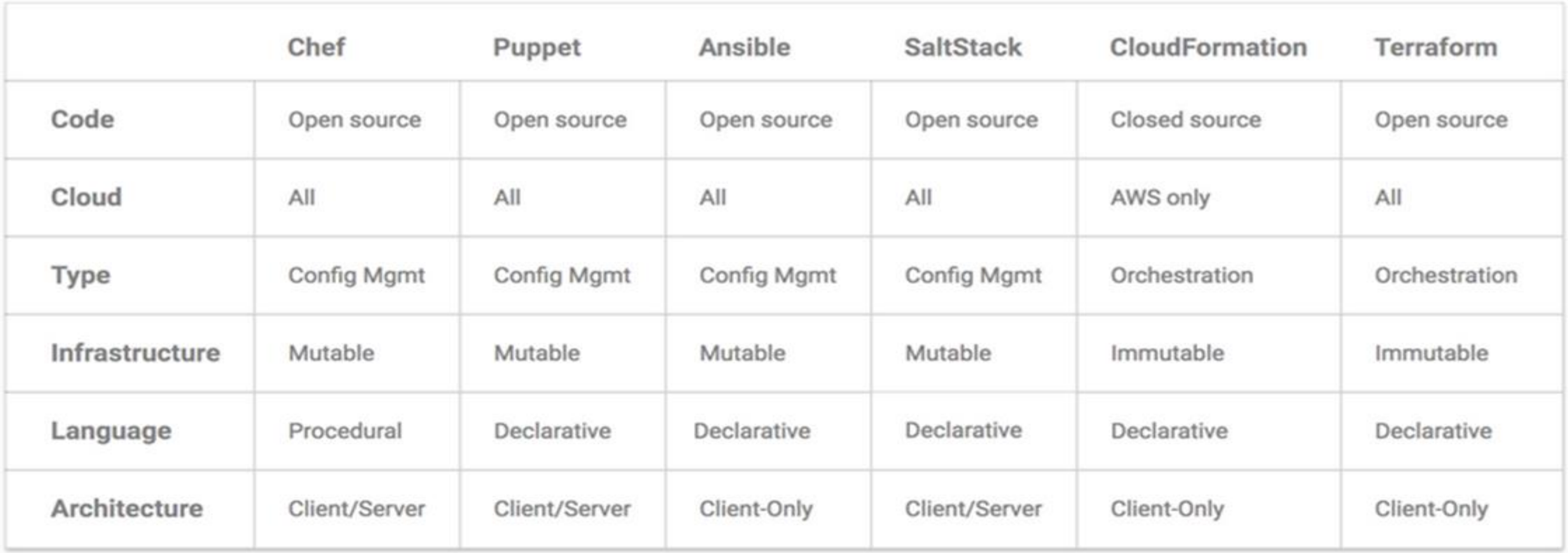
GUI vs CLI vs IAC

- **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

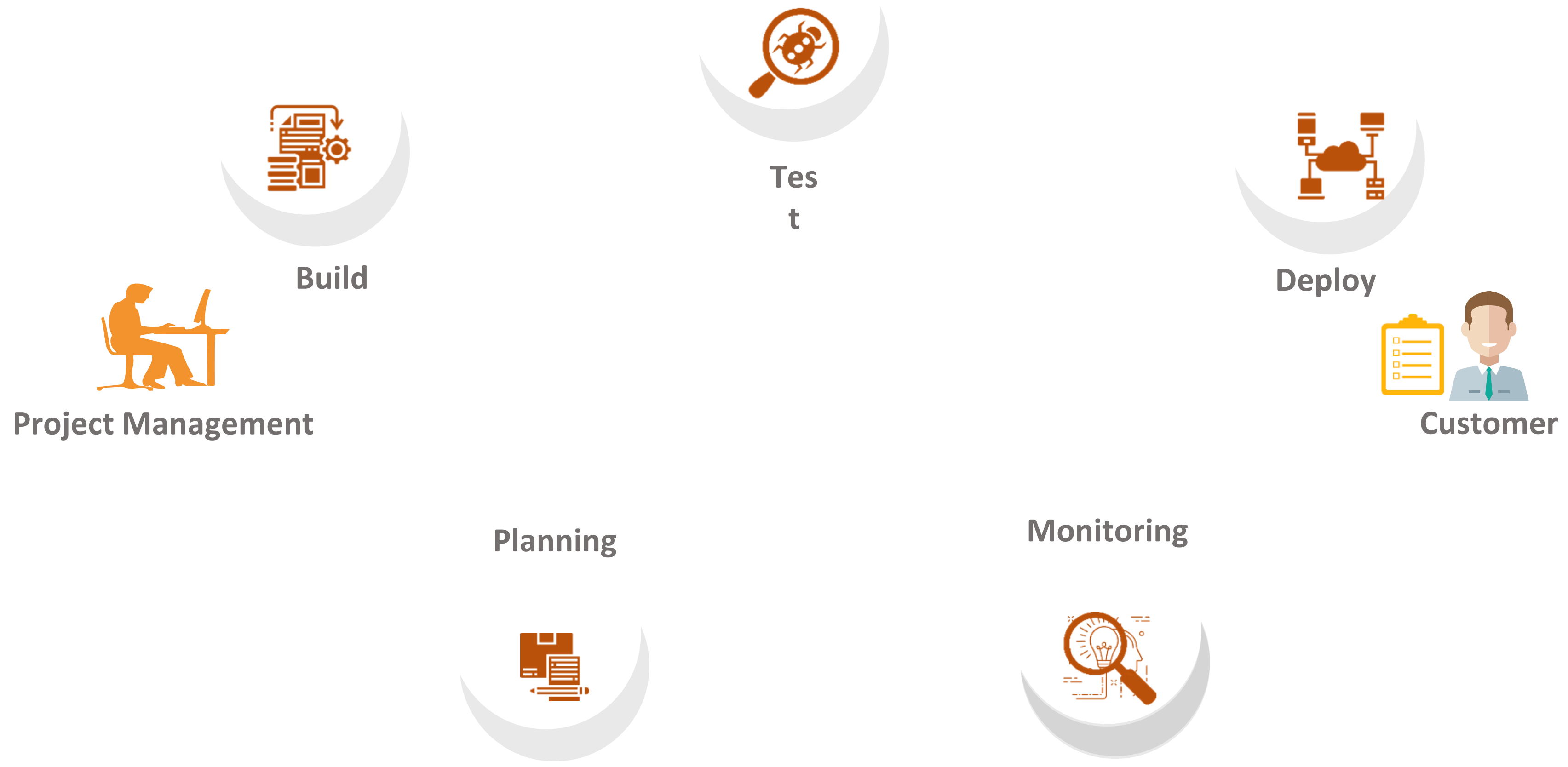
- **IaC (Infrastructure as Code)**
  - Best for Admin Experience
  - Easy management for Admin tasks
  - Easy to understand for end users too
  - Can easily maintain consistency and desired state
  - Infrastructure is written in files, so can be versioned

Terraform and its Peers

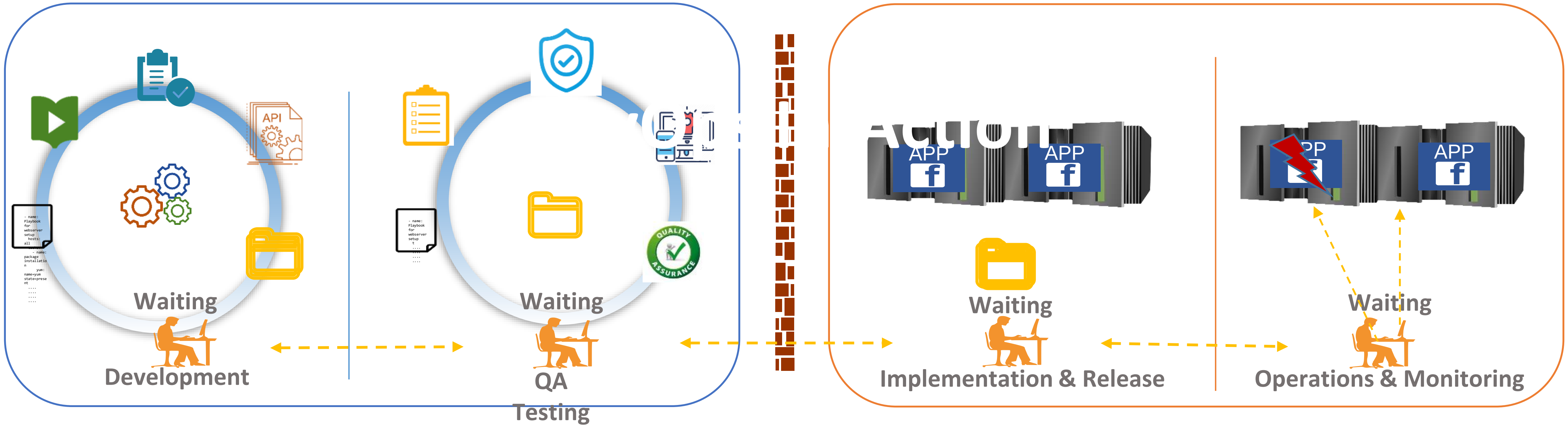


	Chef	Puppet	Ansible	SaltStack	CloudFormation	Terraform
Code	Open source	Open source	Open source	Open source	Closed source	Open source
Cloud	All	All	All	All	AWS only	All
Type	Config Mgmt	Config Mgmt	Config Mgmt	Config Mgmt	Orchestration	Orchestration
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable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

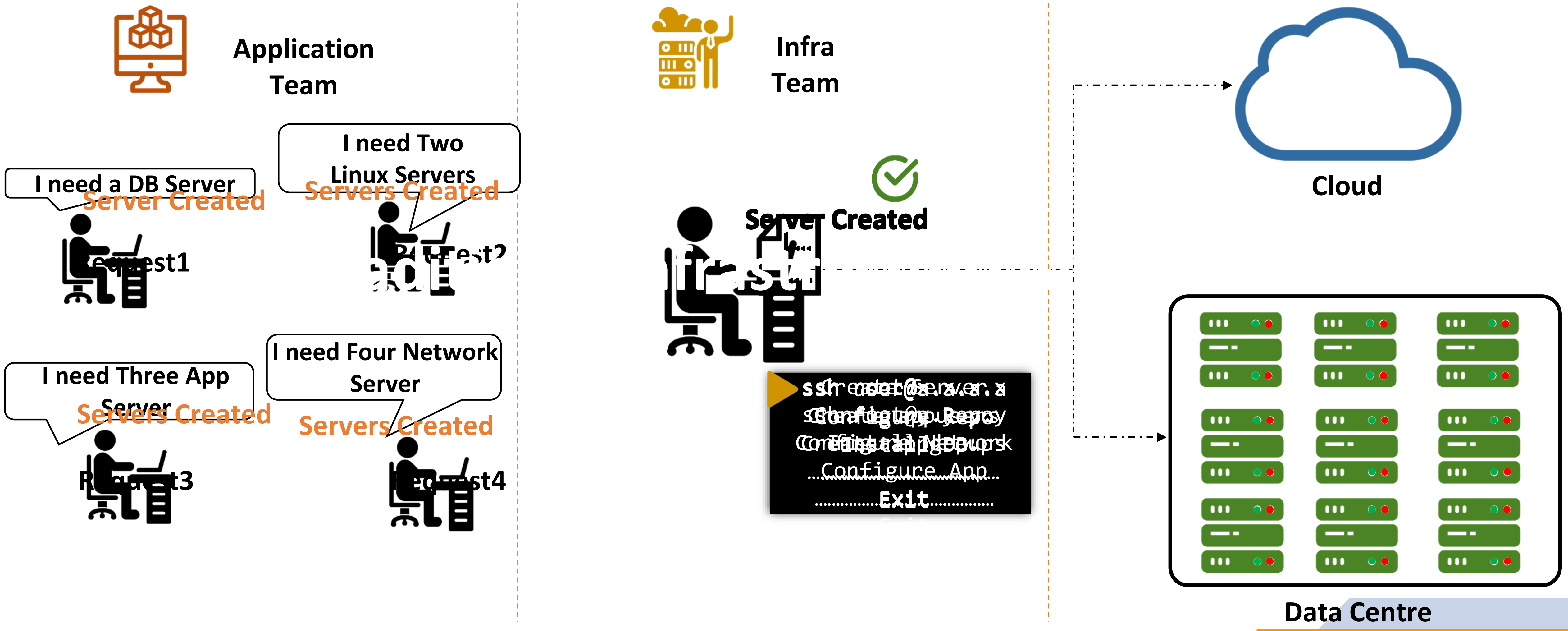
DevOps


Tes
t
Build
Deploy
Project Management
Customer
Planning
Monitoring

DevOps in Action



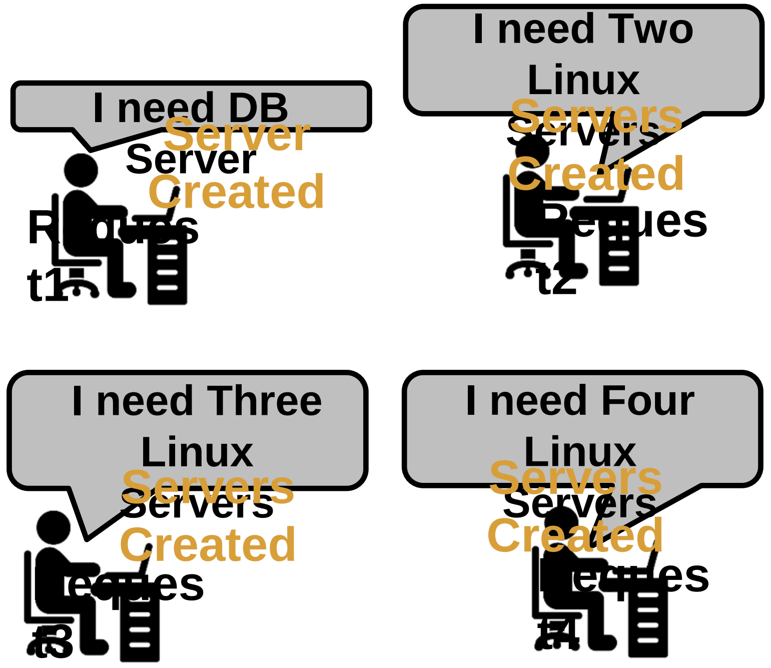
Why DevOps IaC



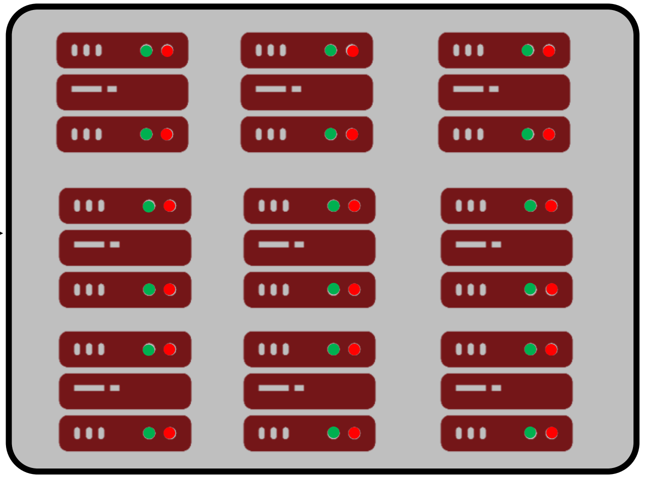
```
ssh user@x.x.x.x
Create Server
ssh root@x.x.x.x
Configure Proxy
Configure Repo
Install Network
Create app Groups
Configure App
Exit
```

Application Team

Infra



```
File is: main.tf
provider "aws" {
  region = "us-east-1"
}
resource "aws_instance" “requestfour" {
  count = "4"
  ami = "ami-030ff268bd7b4e8b5"
  instance_type = "t2.micro"
  tags = {
    Name = "DevOpsInAction"
  }
}
output "myawsserver" {
  value =
"${aws_instance.myawsserver.public_ip}"
}
```



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)

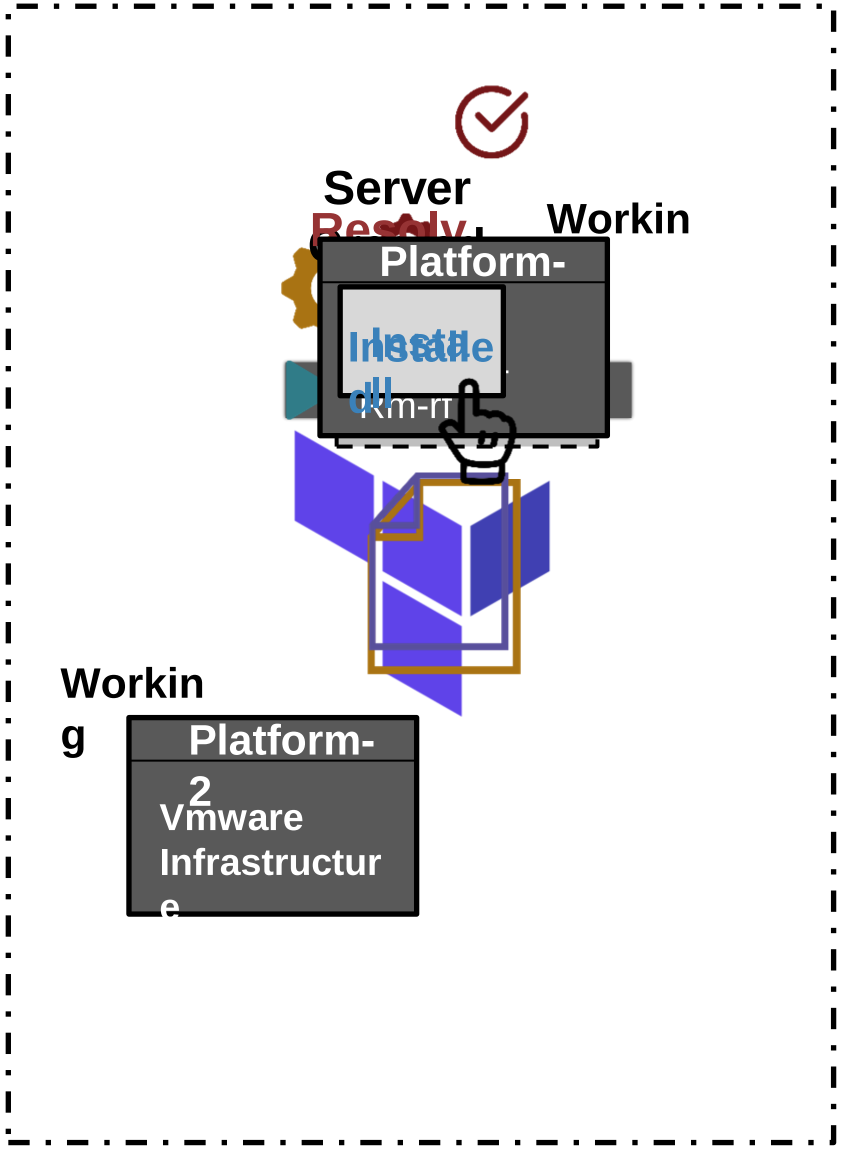
Jul 2014

July 2014, HashiCorp

Opensource / Enterprise

HCL (Hashicorp Configuration Language)

Terraform



ature &
vantages

Easy Installation

Declarative in Nature

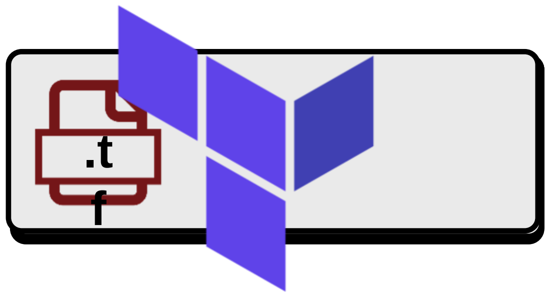
Idempotent

Intelligent Dependency Resolver

Platform Agnostic

Simple and easy to use

Terraform



Provider
s

Variabl
es

Resourc
es

Provisione
rs

DataSourc
es

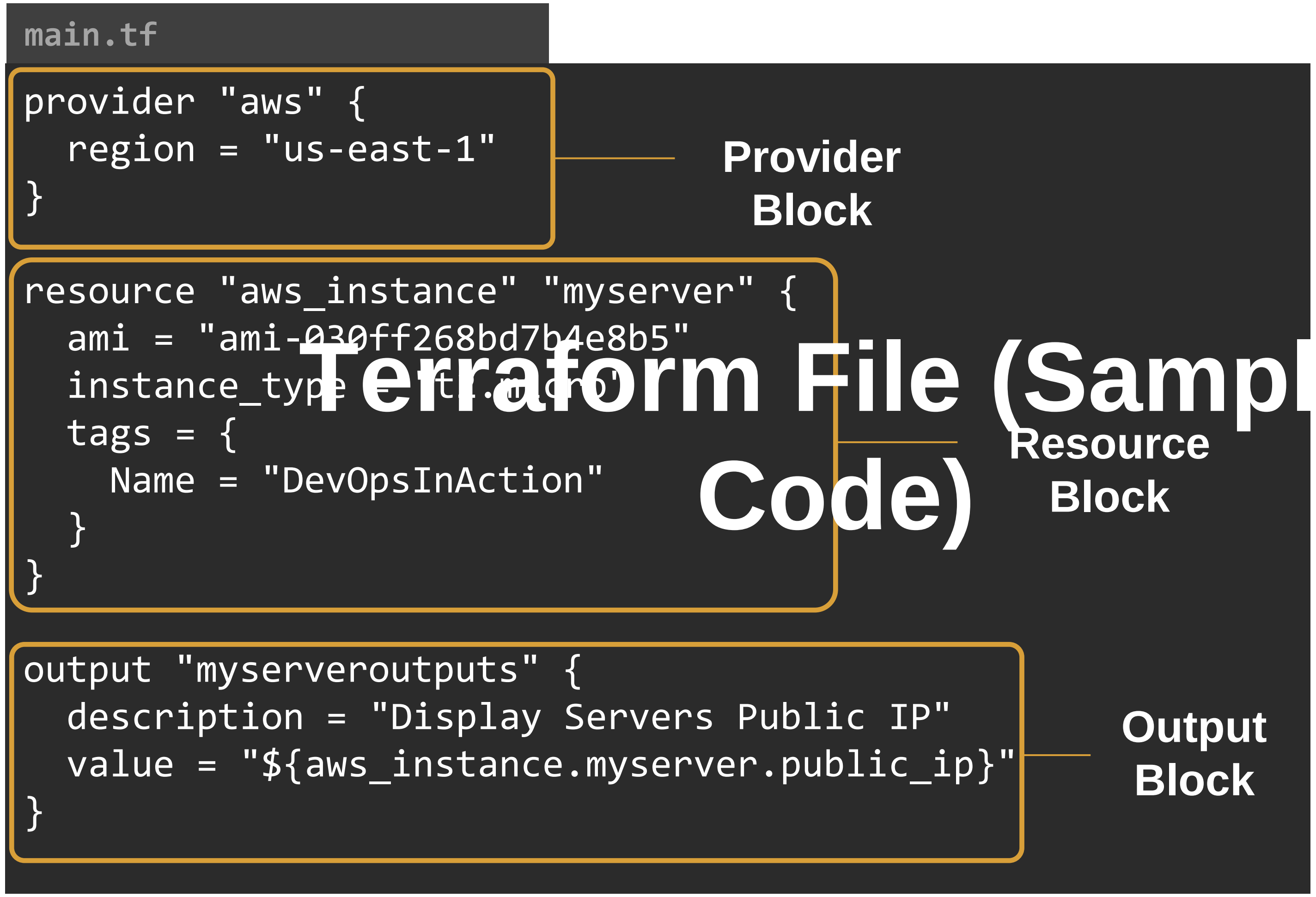
Output
s

Module
s

File extension
.tf

Terraform

Terraform File (Sampl Code)



main.tf

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

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

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

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
- Client/Server Architecture vs Client-Only Architecture

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.

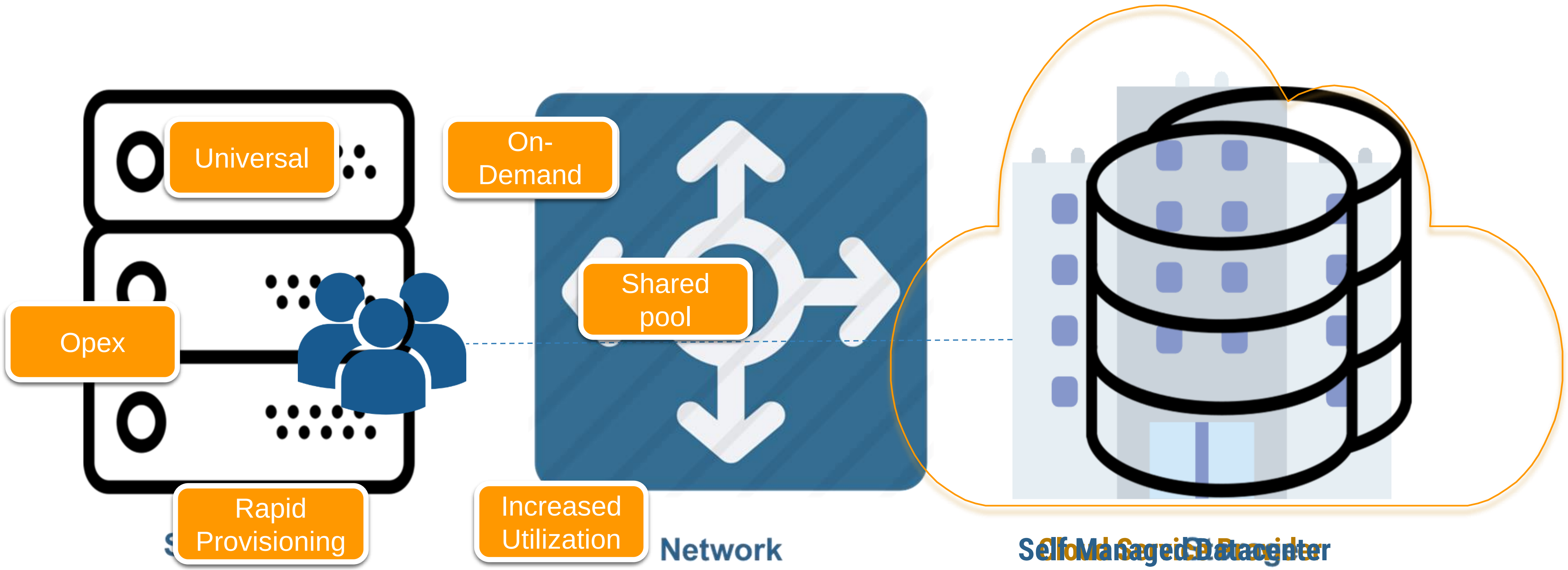
- ➢ Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- ➢ Terraform can manage existing and popular service providers as well as custom in-house solutions.
- ➢ Maintain Desired State
- ➢ Highly scalable and can create a complete datacenters in minutes
- ➢ 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.

Cloud

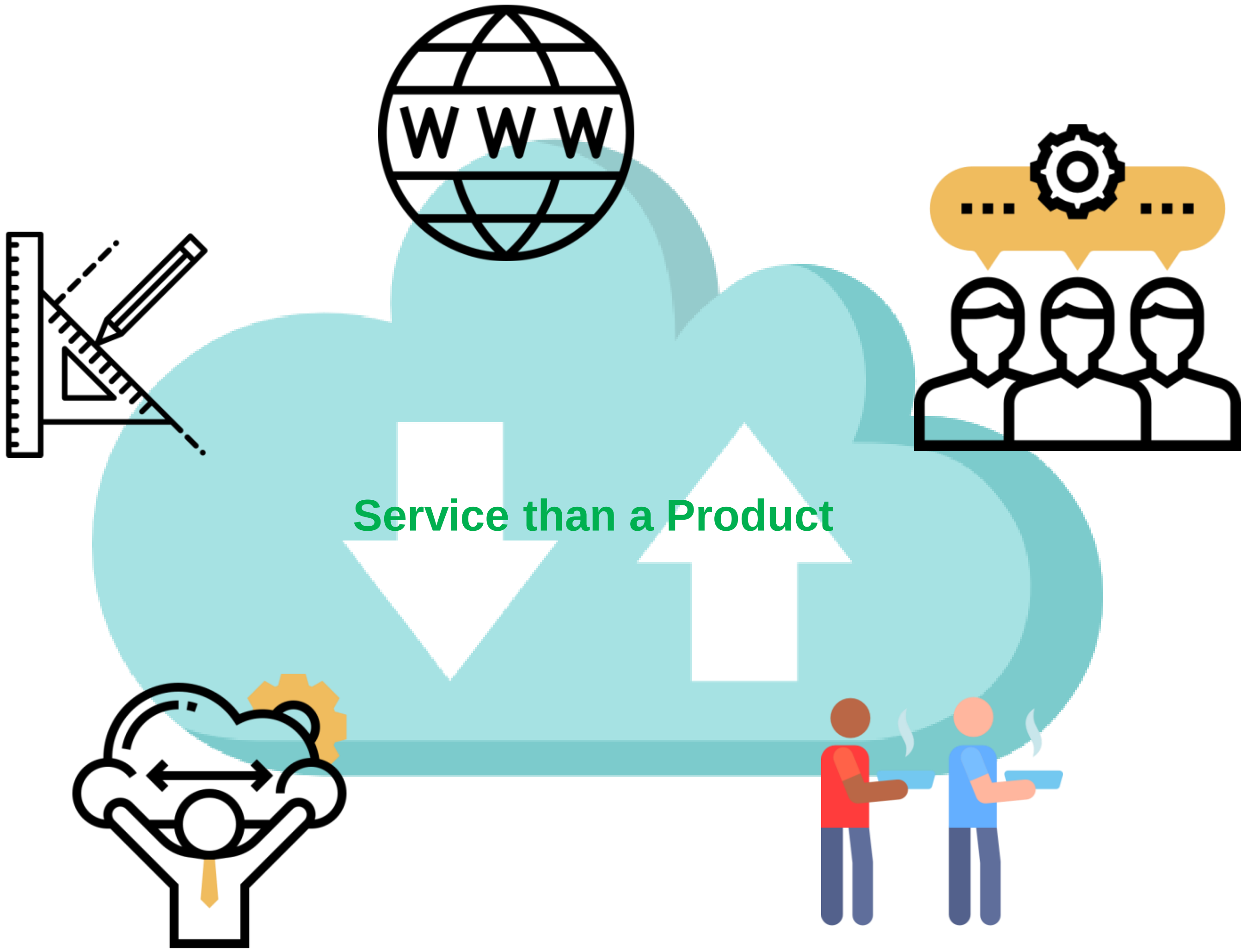
What is Cloud?

Traditional Datacenter

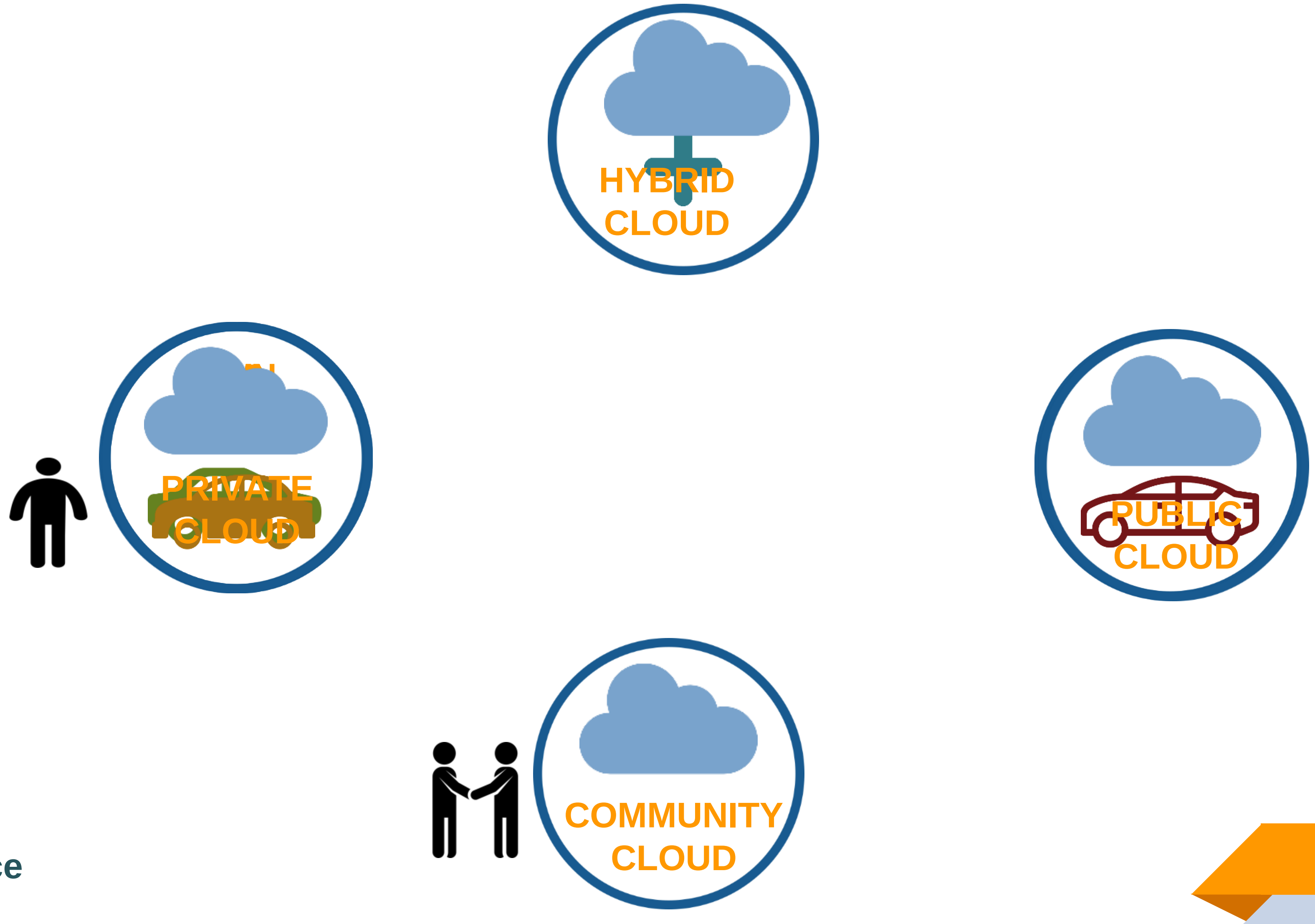
CLOUD



CHARCTERSTICS OF CLOUD



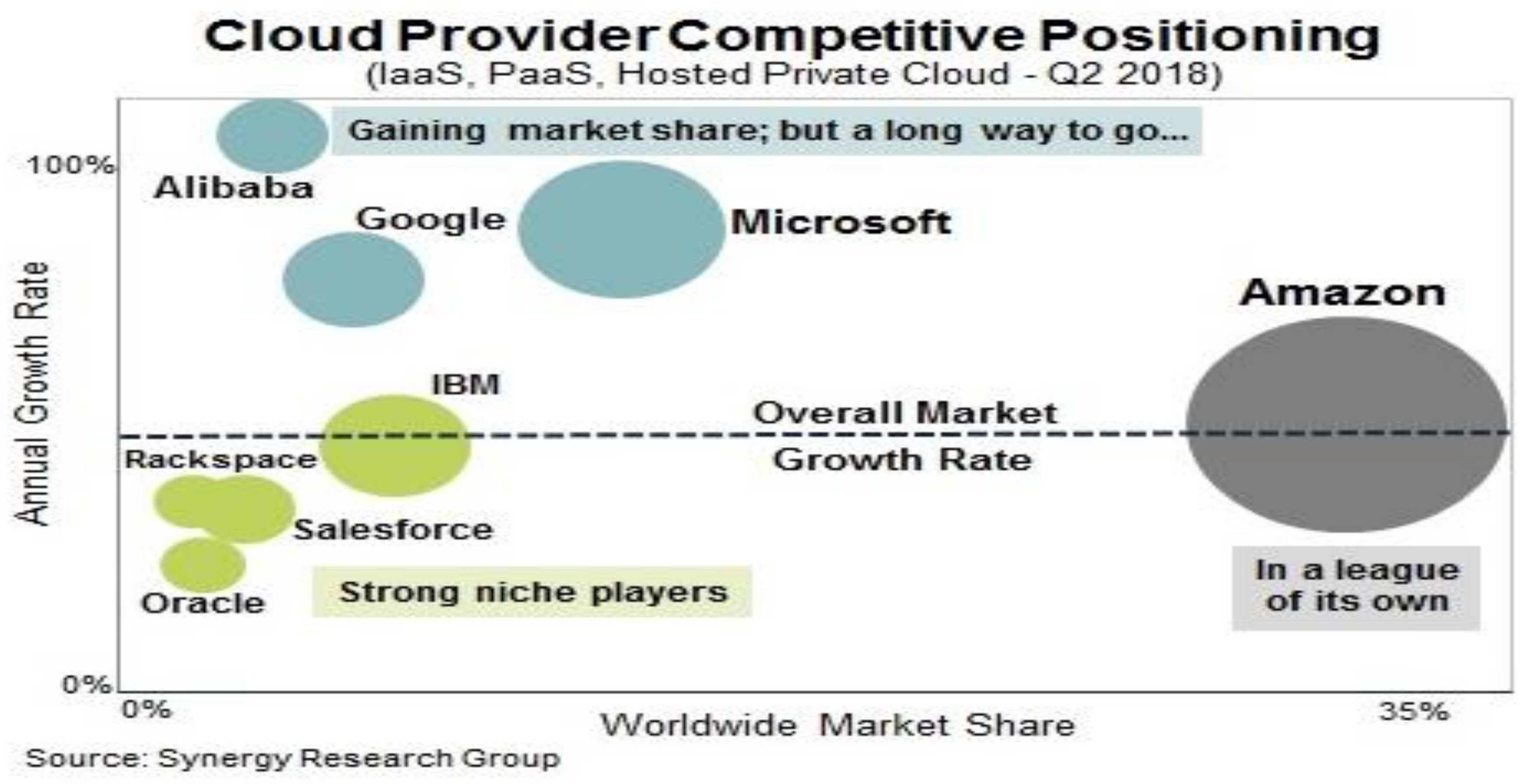
CLOUD DEPLOYMENT TYPE



Transport Service

Cloud Service

CLOUD PLAYERS



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.

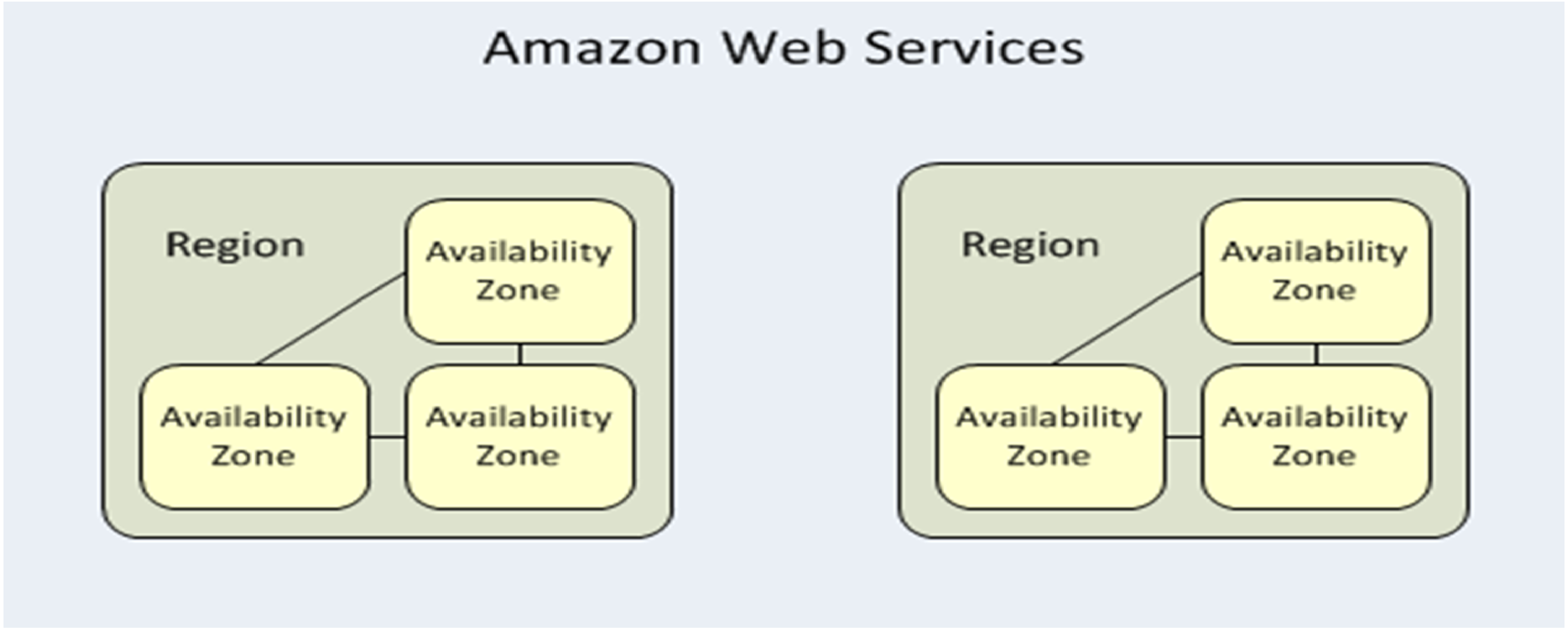
Scalability: Facing some resource issues, no problem within seconds you can scale up the resources and improve your application performance. This cannot be compared with traditional IT datacenters.

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.

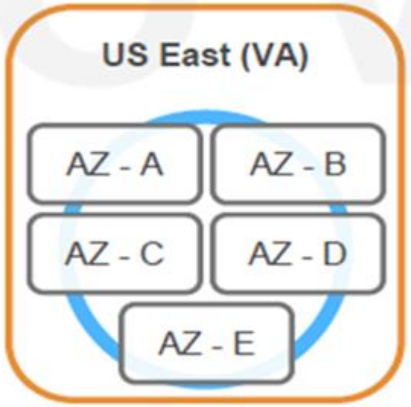
Amazon Web Services

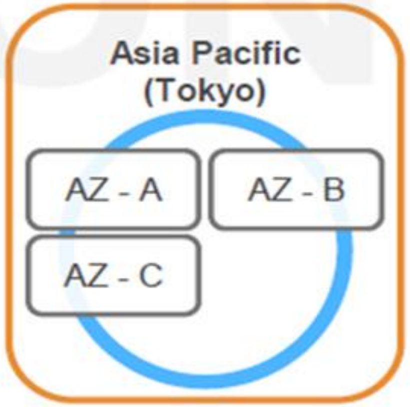


Amazon Web Services

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:

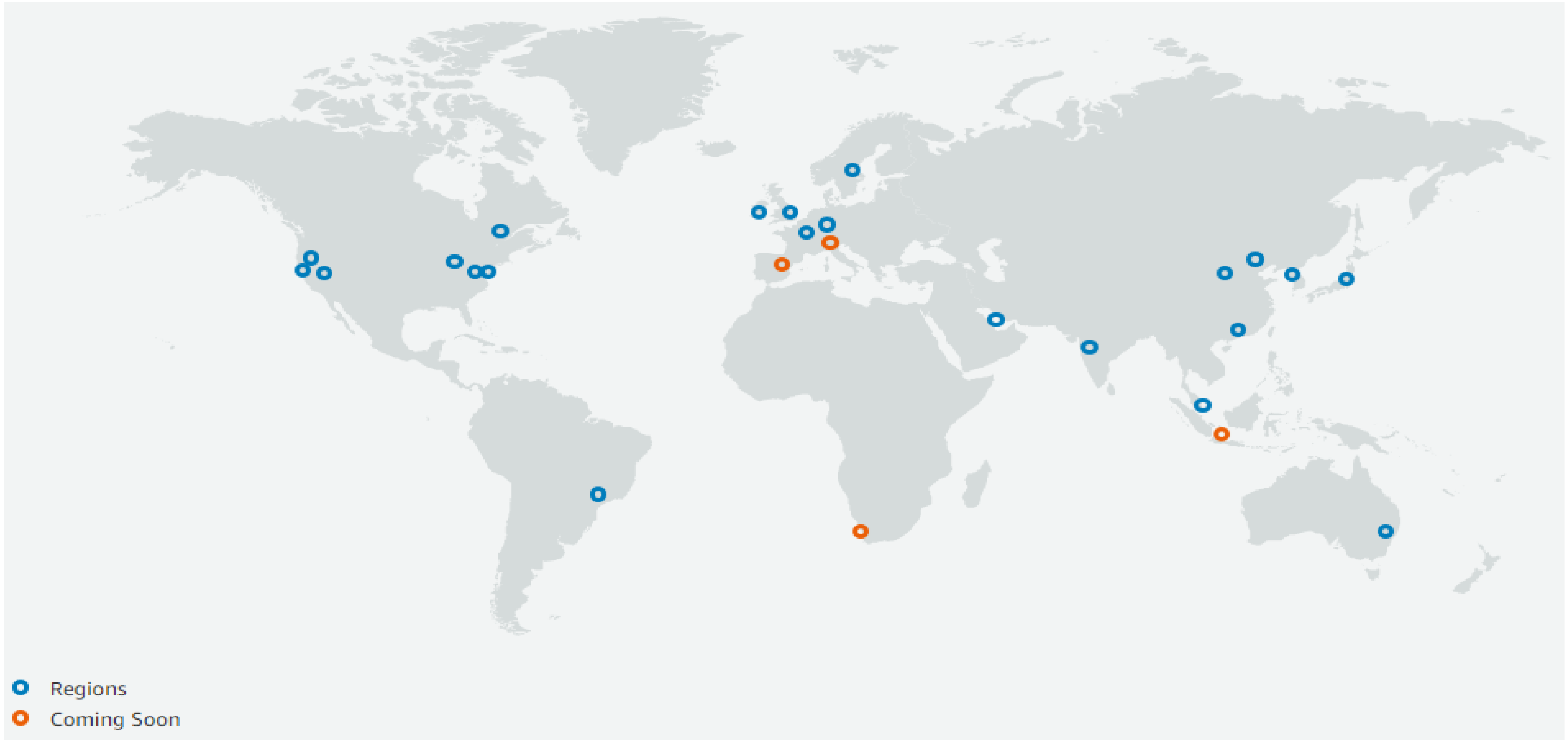
22 AWS Regions

69 AZs

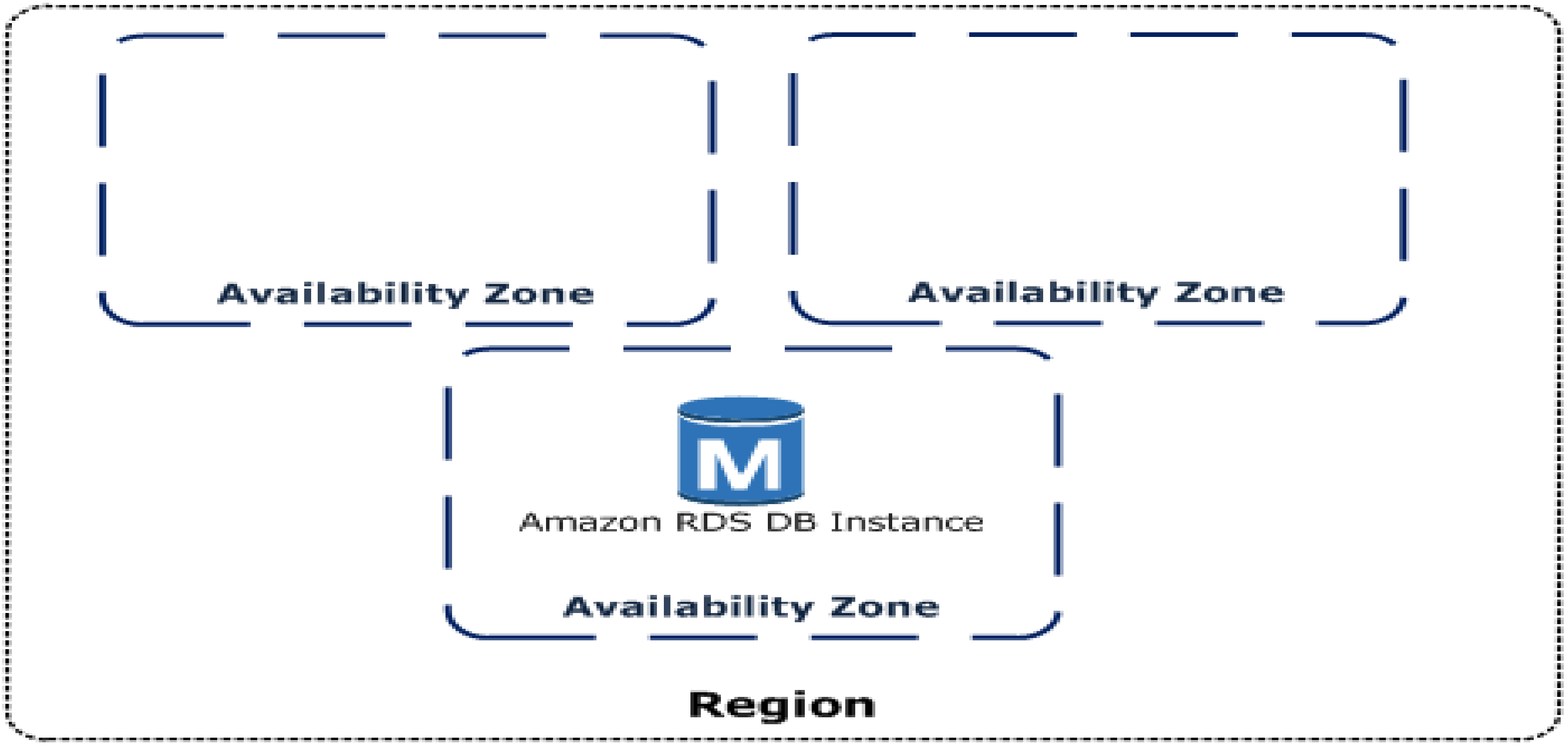
Upcoming:

4 Regions

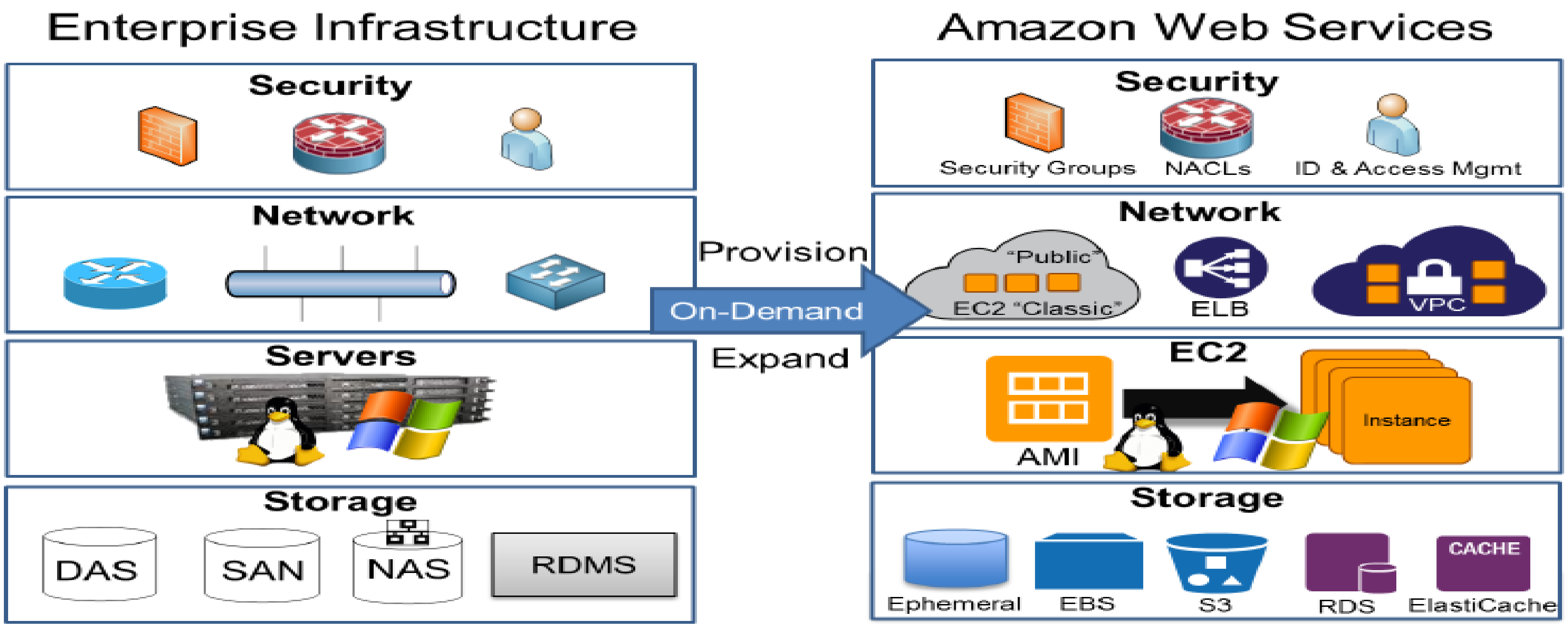
13 AZs



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

Knowledge Checks

- To Achieve HA in AWS, my Servers should be in different (Racks, Datacenters, Availability Zones, Regions)?

- To Achieve DR/High Durability, where should I have by Server backup (Different AZ or Different Regions)?

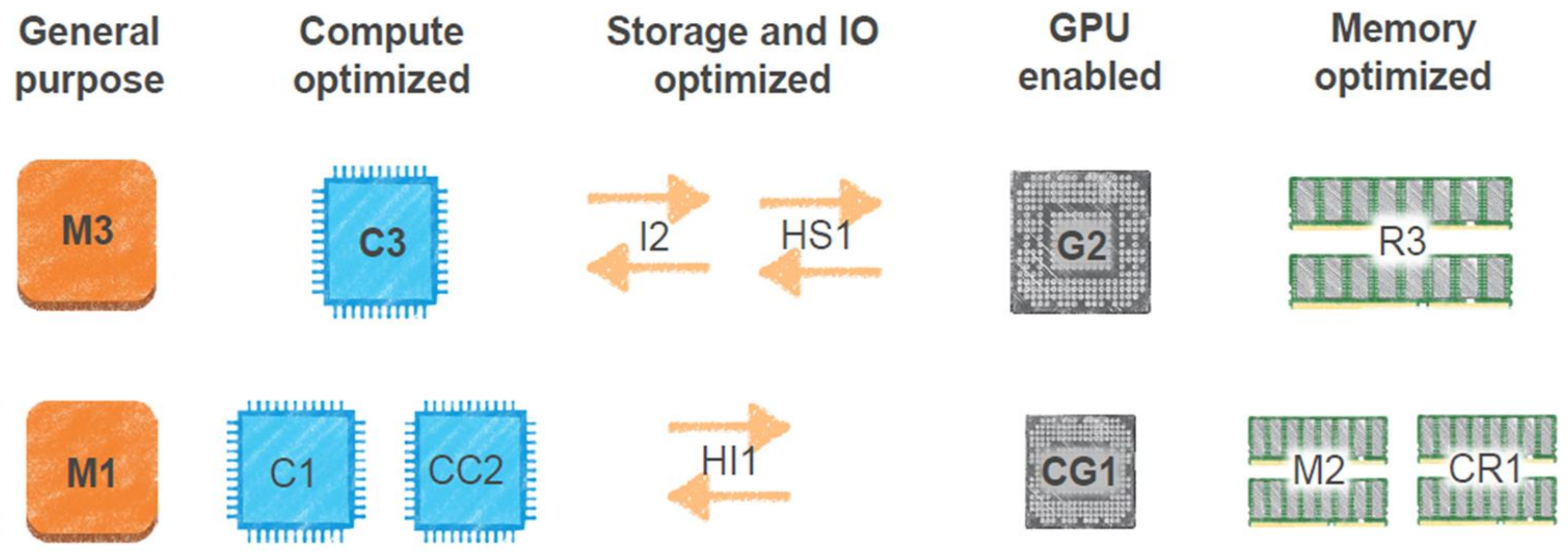
AWS
Compute Services

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.

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:~#
```

AWS CLI

Check the details for all running instances using CLI

- aws ec2 describe-instances

Creation of an AWS Instance using CLI:

- aws ec2 run-instances help
- aws ec2 run-instances --image-id ami-76d6f519 --instance-type t2.micro --key-name test
- aws ec2 describe-instances
- aws ec2 stop-instances --instance-ids i-02e6b6c6c4dd3bbe0
- aws ec2 terminate-instances --instance-ids i-0297acea9e1b39a56

Microsoft Azure

Microsoft Azure

Azure is a cloud offering from Microsoft that individual and organizations can use to create, deploy and operate Cloud based apps and Infrastructure services.

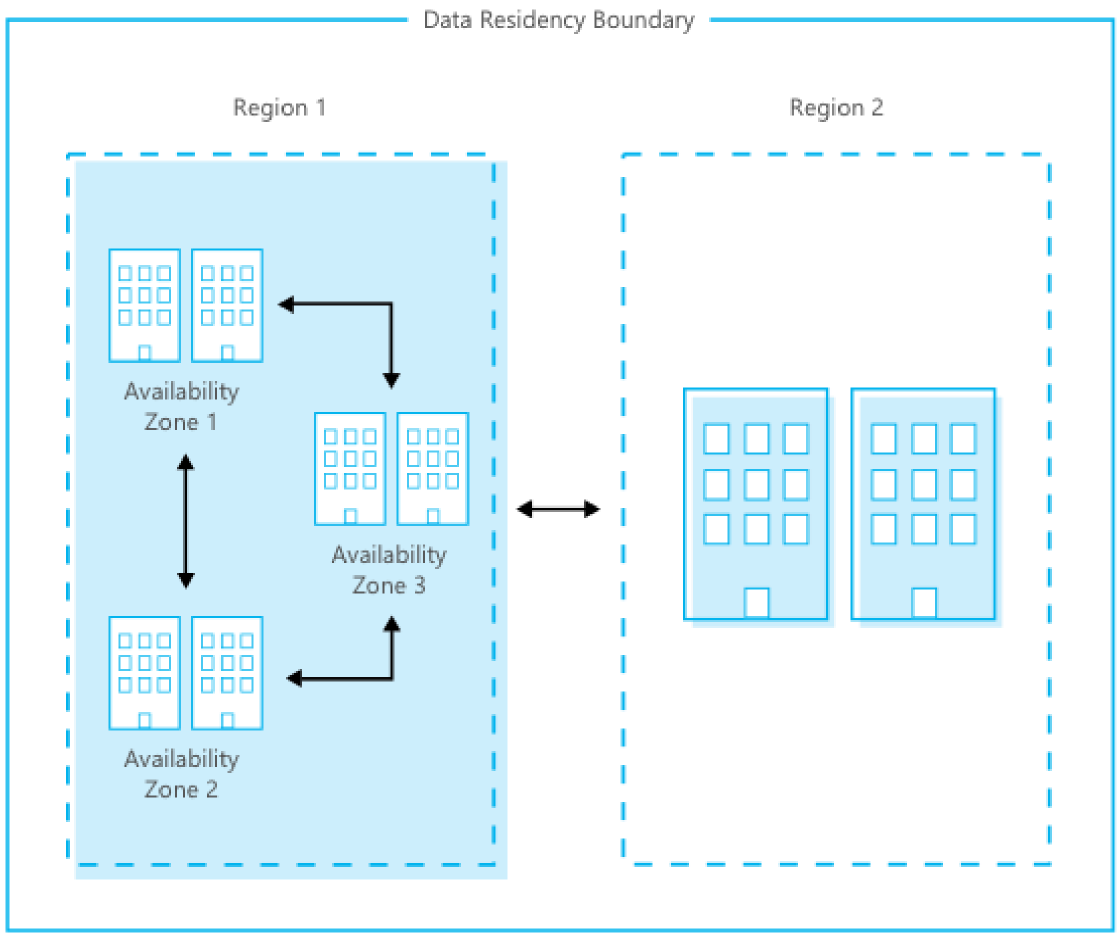
- Best in class support services offered by Microsoft since its release in 2010.
- Available for purchase in 140 countries around the world and support billing in 24 currencies
- Major Focus on PaaS and SaaS.
- 54 Azure regions (42 Current, 12 planned) – more than any cloud provider (Q4-2019)
- 90+ Compliance offerings – again more than any cloud provider as of now (Q4-2019)

Service Locations

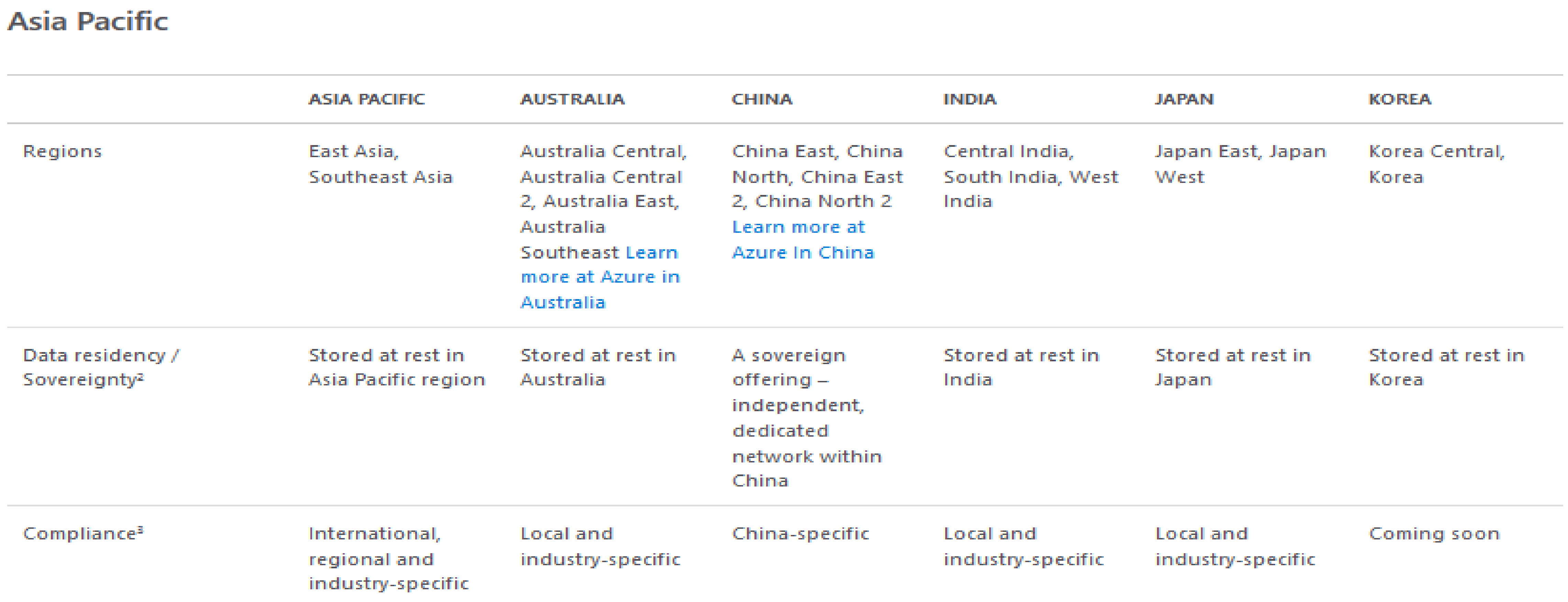
Availability Zones
Availability Zones are physically separate locations within an Azure region. Each Availability Zone is made up of one or more datacenters equipped with independent power, cooling and networking.

Regions
A region is a set of datacenters deployed within a latency-defined perimeter and connected through a dedicated regional low-latency network. Up to 1.6 Pbps of bandwidth in a region

Geographies
A geography is a discrete market, typically containing two or more regions, that preserves data residency and compliance boundaries. Geographies allow customers with specific data-residency and compliance needs to keep their data and applications close.



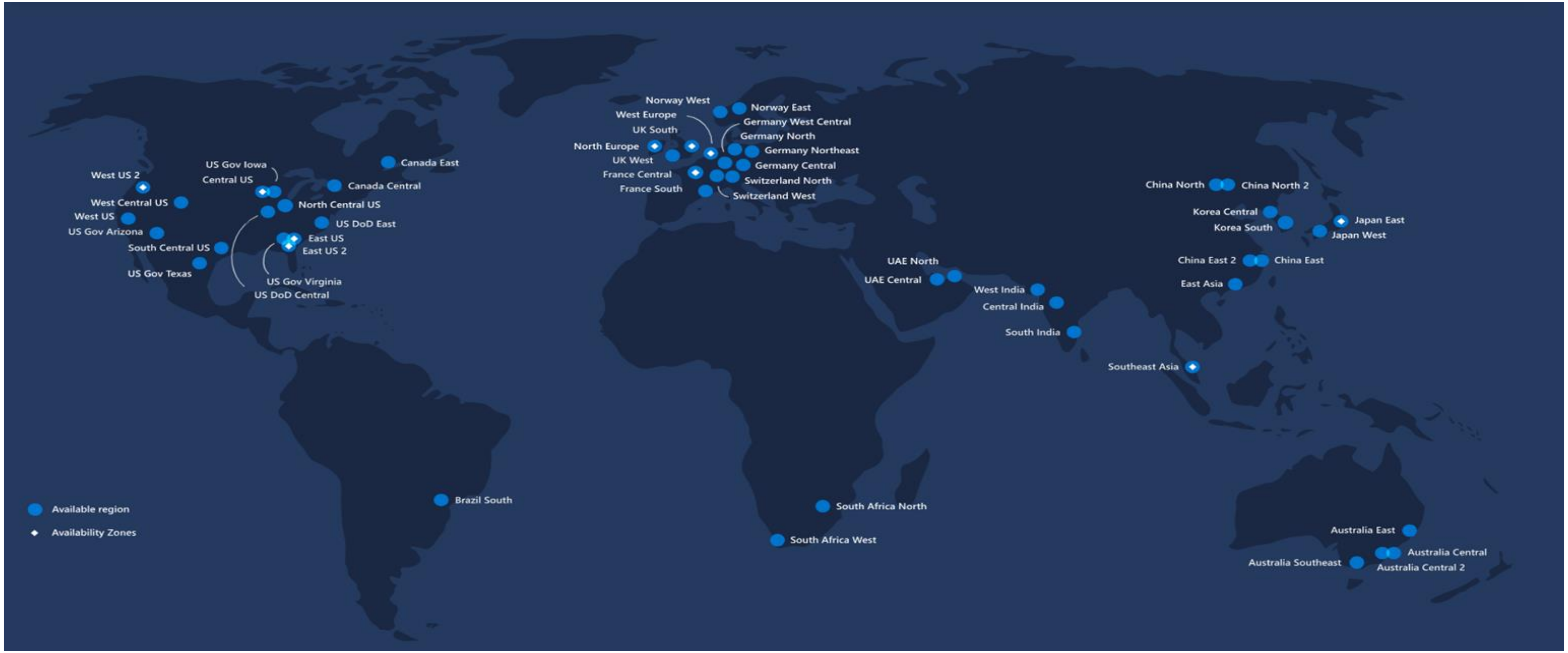
Service Locations



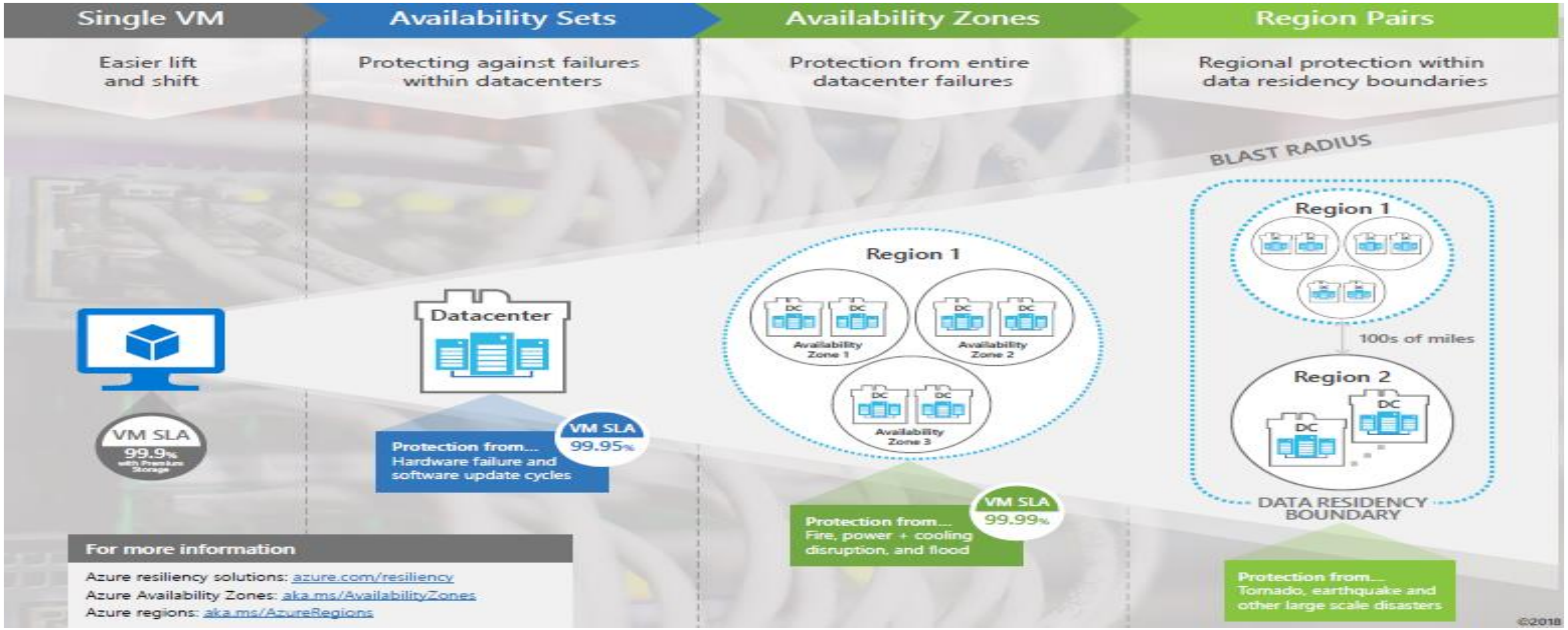
Asia Pacific

	ASIA PACIFIC	AUSTRALIA	CHINA	INDIA	JAPAN	KOREA
Regions	East Asia, Southeast Asia	Australia Central, Australia Central 2, Australia East, Australia Southeast Learn more at Azure in Australia	China East, China North, China East 2, China North 2 Learn more at Azure In China	Central India, South India, West India	Japan East, Japan West	Korea Central, Korea
Data residency / Sovereignty[2]	Stored at rest in Asia Pacific region	Stored at rest in Australia	A sovereign offering – independent, dedicated network within China	Stored at rest in India	Stored at rest in Japan	Stored at rest in Korea
Compliance[3]	International, regional and industry-specific	Local and industry-specific	China-specific	Local and industry-specific	Local and industry-specific	Coming soon

Service Locations



Service Locations

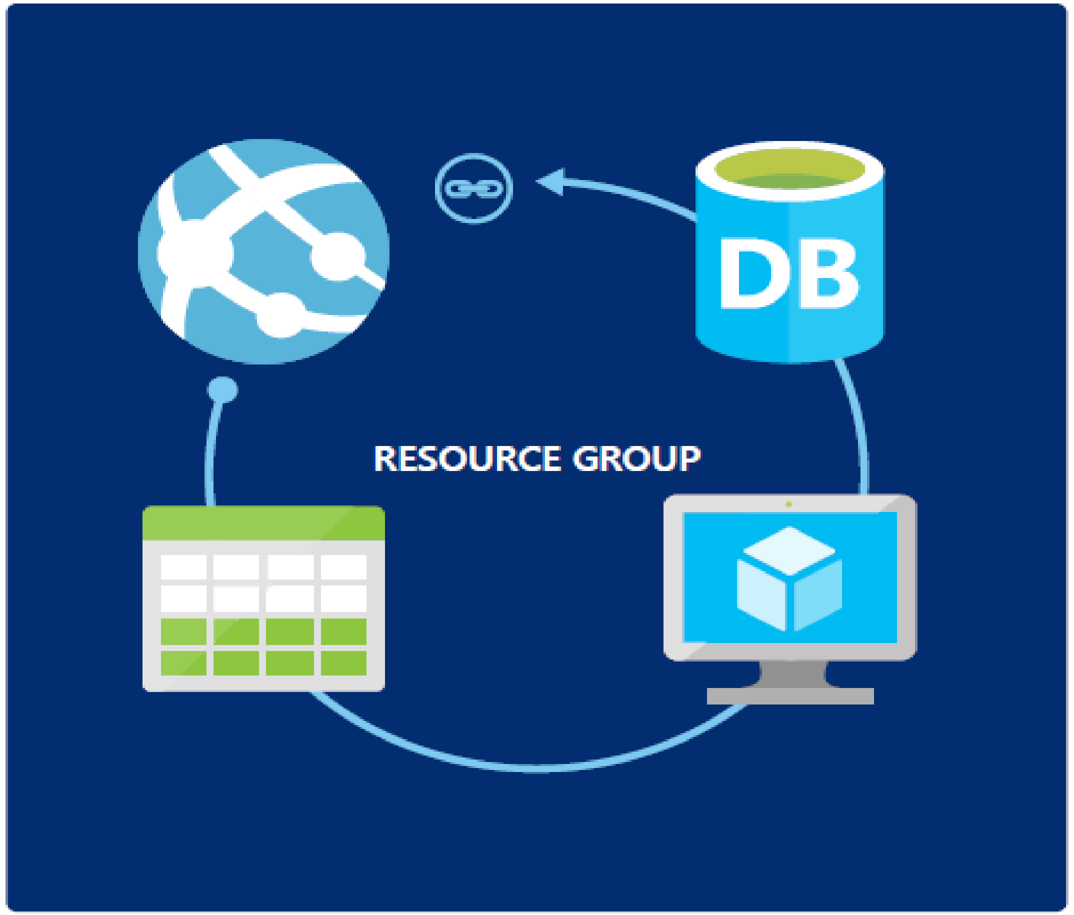


Single VM
Easier lift and shift
Availability Sets
Protecting against failures within datacenters
Availability Zones
Protection from entire datacenter failures
Region Pairs
Regional protection within data residency boundaries
BLAST RADIUS
VM SLA 99.9% with Premium Storage
Datacenter
Protection from... Hardware failure and software update cycles
VM SLA 99.95%
Region 1
DC
Availability Zone 1
Availability Zone 2
Availability Zone 3
Protection from... Fire, power + cooling disruption, and flood
VM SLA 99.99%
Region 1
100s of miles
Region 2
DATA RESIDENCY BOUNDARY
Protection from... Tornado, earthquake and other large scale disasters
©2018

For more information

Azure resiliency solutions: azure.com/resiliency
Azure Availability Zones: aka.ms/AvailabilityZones
Azure regions: aka.ms/AzureRegions

Resource Group



- Tightly coupled containers of multiple resources of similar or different types
- Resource groups can span regions
- Every resource *must* exist in one and only one resource group

Resource Group Lifecycle

Question:

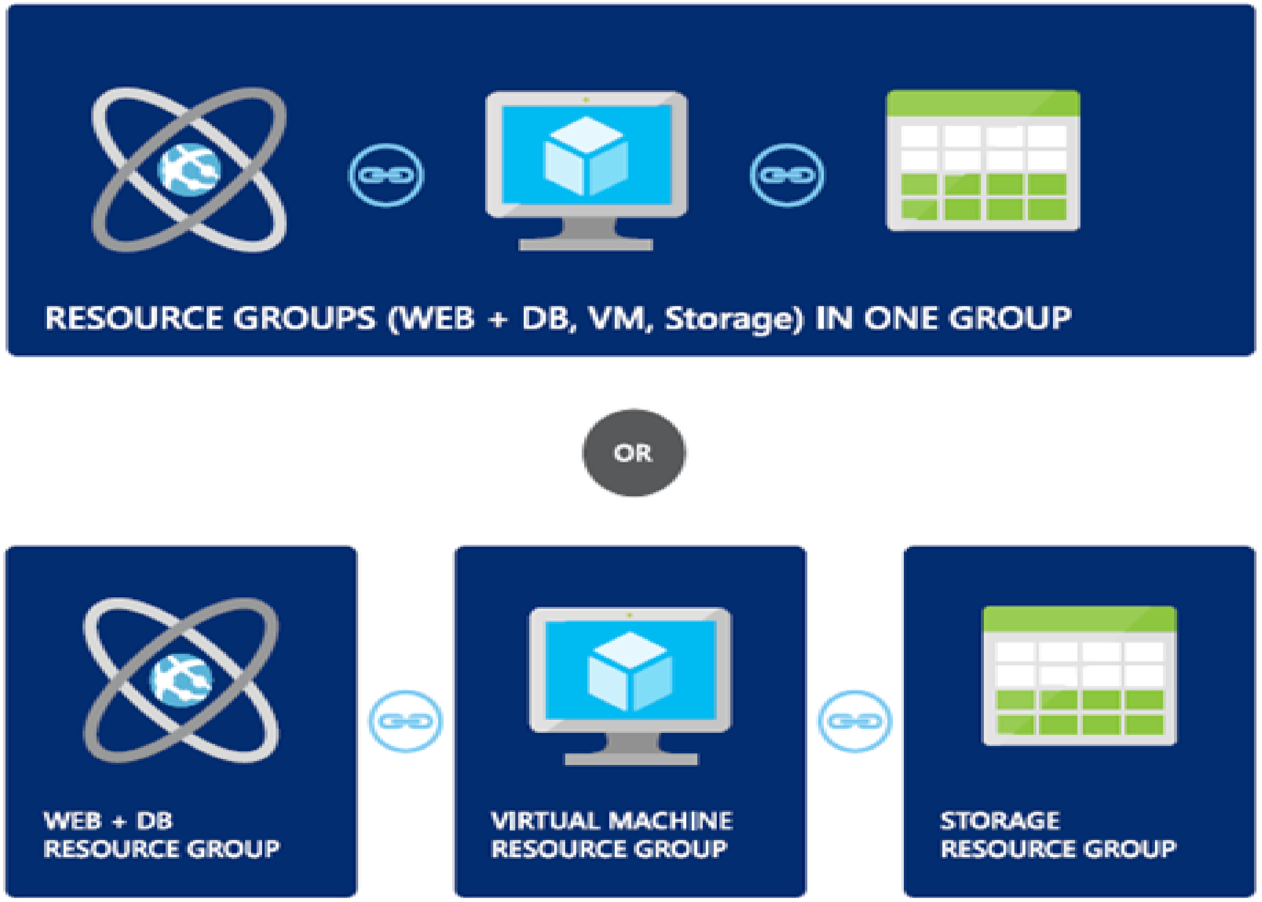
Should these resources be in the same group or a different one?

Hint:

Do they have common lifecycle and management?

Answer:

It's up to you.


RESOURCE GROUPS (WEB + DB, VM, Storage) IN ONE GROUP
OR
WEB + DB RESOURCE GROUP
VIRTUAL MACHINE RESOURCE GROUP
STORAGE RESOURCE GROUP

Accounts and Subscriptions

An Azure account determines how and to whom your Azure usage is reported.

A subscription helps you to organize your access to your cloud services and resources, and also to control how your resource usage is reported, billed and paid for.

Each of your subscriptions can have a different billing and payment setup. As a result, you can have different subscriptions and different plans based on criteria such as department, project, or regional location.

Every Azure resource belongs to a subscription.

Microsoft Azure VMs

Azure VM

Azure Virtual Machines supports the deployment of Windows or Linux virtual machines (VMs) in a Microsoft Azure datacenter. You have total control over the configuration of the VM. You are responsible for all server software installation, configuration, and maintenance and for operating system patches.

Because of the level of control afforded to the user and the use of durable disks, VMs are ideal for a wide range of server workloads that do not fit into a PaaS model.

Microsoft offers a **99.95 percent connectivity service level agreement (SLA) for multiple-instance VMs deployed in an availability set**. That means that for the SLA to apply, there must be at least two instances of the VM deployed within an availability set.

Network Security Group

Security Group protection for your VMs and resources

Can be applied to single VM or to complete subnet

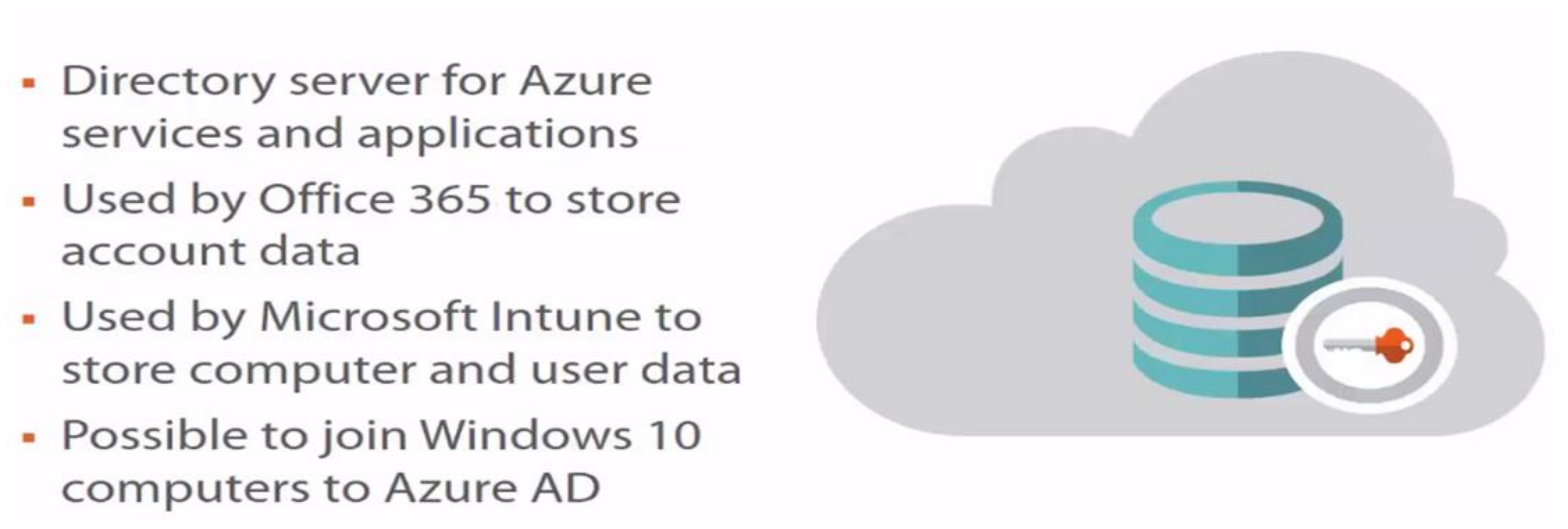
By default all incoming traffic is blocked, all internal traffic between VMs of same network is allowed

By default all outgoing traffic is allowed.

NSGs are actually applied to a NIC attached to a VM (rather than the VM itself). If a VM has multiple NICs, the NSG needs to be applied separately to each NIC.

Azure Active Directory

Azure Active Directory



- Directory server for Azure services and applications
- Used by Office 365 to store account data
- Used by Microsoft Intune to store computer and user data
- Possible to join Windows 10 computers to Azure AD

Azure AD

- Authentication
- Authorization
- RBAC
- Users/Groups/Roles
- System Managed Identities
- Service Principals

Azure AD

- az login
- az ad sp create-for-rbac --role="Contributor" --scope="/subscriptions/18a63478-19db-4208-b4f3-4e007ab700aa"
- az logout
- az login --service-principal -u "235dc2ec-d9e8-42af-b2b4-75acd2876ec0" -p "ZtIeK5IGLDPtZW.PE9Zfuwsy~IX0f6Vn7~" --tenant "cee10f4c-766b-4d55-afcf-eebf78bd9f61“
- az account show
- az account list --query "[].{name:name, subscriptionId:id}"
- export ARM_CLIENT_ID="235dc2ec-d9e8-42af-b2b4-75ac876ec0"
- export ARM_CLIENT_SECRET="ZtIeK5IGLDPtZW.PE9Zfuwsy~If6Vn7~“
- export ARM_SUBSCRIPTION_ID="18a63478-19db-408-b4f3-4e007ab700aa"
- export ARM_TENANT_ID="cee10f4c-766b-4d55-acf-eebf78bd9f61"

Azure CLI

Azure CLI

For Windows users, the PowerShell cmdlets suits best. While for mixed environments, the Azure cross-platform command-line interface provides a consistent experience for Linux, Mac OS, and Windows users.

The Azure CLI is a Node.js application implemented by using the Azure SDK for Node.js.

Download Azure CLI for Windows from below URI:
https://aka.ms/InstallAzureCliWindows

CLI Installation in rest OS:
https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest

Azure CLI

- az login
- az configure --defaults group=newrg web=myweb vm=myvm location=southindia
- az group create --name newrg -l southindia
- az vm[or any command]   //to see all options wrt same
- az vm image list   // to list vm images
- Az vm image list –publisher CoreOS
- az vm create -n MyVm -g newrg --image UbuntuLTS --authentication-type password --admin-username admnuser --admin-password “anypasswd@123”
- az network vnet create -n MyVirtualNetwork -g MyResourceGroup --address-prefix 10.0.0.0/16
- az network lb create -n MyLoadBalancer -g MyResourceGroup

Terraform Installation

Installation of Terraform

- Terraform is distributed as a single binary. Install Terraform by unzipping it and moving it to a directory included in your system's PATH

- Current Version of Terraform: 0.13.2

- Download the latest version: https://www.terraform.io/downloads.html

- You can verify the checksum at https://releases.hashicorp.com/terraform/0.13.2/terraform_0.13.2_SHA256SUMS

- Older versions can be downloaded from: https://releases.hashicorp.com/terraform/

Installation of Terraform

- Check SHA256SUM of your downloaded zip file (Mandatory for Production Security):

```
[root@TechLanders ~]# sha256sum terraform_0.13.0_linux_amd64.zip
 9ed437560faf084c18716e289ea712c784a514bdd7f2796549c735d439dbe378  terraform_0.13.0_linux_amd64.zip
 [root@TechLanders ~]#
```

- Unzip your downloaded file and copy the terraform executable binary to your /usr/bin or set the environment variable for executable path.

- Run Terraform –version to check the installation and integrity of binary:

```
 [root@TechLanders ~]# terraform -version
 Terraform v0.13.0
 [root@TechLanders ~]#
```

Installation of Terraform

```
[root@TechLanders ~]# terraform -version
Terraform v0.13.0
[root@TechLanders ~]#

[root@TechLanders ~]# terraform --help
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply         Builds or changes infrastructure
    console       Interactive console for Terraform interpolations
    destroy       Terraform-managed infrastructure
    env           Workspace management
```

Installation of Terraform

- Sub-command help can be taken using –help with subcommand:

```
[root@ip-172-31-6-233 ~]# terraform -help plan
Usage: terraform plan [options] [DIR]
Generates an execution plan for Terraform.
Options:
  -compact-warnings   If Terraform produces any warnings that are not
                      accompanied by errors, show them in a more compact form
                      that includes only the summary messages.
```

- To install auto-complete for sub-commands(for bash and zsh), enable auto-complete:

```
[root@ip-172-31-6-233 ~]# terraform -install-autocomplete
[root@ip-172-31-6-233 ~]# terraform    //double-tab
0.12upgrade    debug      force-unlock   init             output       refresh    untaint
0.13upgrade    destroy    get            internal-plugin  plan         show       validate
apply          env        graph          login            providers    state      version
console        fmt        import         logout           push         taint      workspace
[root@ip-172-31-6-233 ~]# terraform
```

Terraform Upgrade

- You have to relook at your Terraform configuration file (HCL) in case you are changing the Terraform version, as there are several enhancements, bugfixes and older commands deprecations.
- Changes of the latest release can be found at :
  https://github.com/hashicorp/terraform/blob/master/CHANGELOG.md
- Changes (Enhancements and Bug fixes) of previous releases can be found at same link with Version number instead of master: https://github.com/hashicorp/terraform/blob/v0.12/CHANGELOG.md
- Terraform supports upgrade tools and features only for one major release upgrade at a time.
- For errors and issue, you can raise a case at Terraform community forum - https://discuss.hashicorp.com/

Terraform Fundamentals

Providers

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

Terraform automatically discovers provider requirements from your configuration, including providers used in child modules.

To see the requirements and constraints, run "terraform providers".

```
C:\Users\gagandeep\terra>terraform providers
Providers required by configuration:
.
└── provider[registry.terraform.io/hashicorp/aws]
C:\Users\gagandeep\terra>
```

A provider is responsible for creating and managing resources.

Multiple provider blocks can exist if a Terraform configuration manages resources from different providers.

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, or higher-level components such as DNS records.

```
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.

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.

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 = "AKIAJB2KQBDLH56XQEYA"
  secret_key = "rNNWWuzvBpp+v//OXCB10Zr2OVuPI3iayxXXStPs"
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"

  tags = {
    Name = "Techlanders-aws-ec2-instance"
  }
}

output "myawsserver" {
  value = "${aws_instance.myawsserver.public_ip}"
}
```

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 validate to validate the template
4) Run terraform init to initialize the plugins and modules
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
----
----
```

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...
google_compute_address.vm_static_ip: Creation complete after 5s [id=projects/accenture-286519/regions/us-central1/addresses/terraform-static-ip]
google_compute_instance.vm_instance: 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
The given plan file can no longer be applied because the state was changed by
another operation after the plan was created.
C:\Users\gagandeep\terra>
```

Idempotency

- Run Terraform apply again and check the status of the server.

```
[root@TechLanders aws]# terraform apply
aws_instance.myserver: Refreshing state... [id=i-0a63756c96d338801]

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
[root@TechLanders aws]#
```

- Stop the server and then check. You'll have no change, as server still exists, its just stopped.
- Run Terraform Plan to check the status:

```
[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]

    No changes. Infrastructure is up-to-date.

    This means that Terraform did not detect any differences between your configuration and real physical resources that exist. As a result, no actions need to be
    performed.

    [root@TechLanders aws]#
```

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>
```

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.
```

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.

```
C:\Users\gagandeep\Desktop\terraform>terraform destroy -target=google_compute_instance.vm_instance2 -auto-approve

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

google_compute_instance.vm_instance2: Refreshing state... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2]

google_compute_instance.vm_instance2: Destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2]

google_compute_instance.vm_instance2: Still destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2, 10s elapsed]

google_compute_instance.vm_instance2: Still destroying... [id=projects/accenture-286519/zones/us-central1-b/instances/terraform-instance2, 20s elapsed]

google_compute_instance.vm_instance2: Destruction complete after 24s

Warning: Resource targeting is in effect

You are creating a plan with the -target option, 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.

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:

```
provider "aws" {
  region = "us-east-2"
 access_key = "AKIAJB2KQH56XQEYA"
  secret_key = "rNNWWuzvBpp+v"
}
resource "aws_instance" "myawsserver" {
  ami = "ami-0a54aef4ef3b5f881"
  instance_type = "t2.small"
  tags = {
    Name = "Techlanders-aws-ec2-instance"
    Env = "Prod"
  }
}
output "myawsserver-ip" {
  value = "${aws_instance.myawsserver.public_ip}"
}
```

Using Resource values

Create a GCP instance with network instance: Add below code to the file:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

Working with change

Modify your terraform file and add tags/labels to it and run terraform plan/apply again:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro“
  tags  = [“web”, “dev”]

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
    }
  }
}
```

Notedown the output of terraform plan stating it’ll be an in-place upgrade

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.

- For example, if you add IP address to a server, reboot will not be required.

```
  network_interface {
    network = google_compute_network.vpc_network.name
    access_config {
     nat_ip = google_compute_address.vm_static_ip.address
    }
  }
}
resource "google_compute_address" "vm_static_ip" {
  name = "terraform-static-ip"
}
```

- On the other side modifying the server size can't be done live. It needs a stop and start of the server. For same you need to grant permission in configuration file:

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "g1-small"
  tags = ["web", "dev", "client1"]
   allow_stopping_for_update = true
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
```

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.

Modify the image type to g1-small in config file and check the output of terraform plan:

```
machine_type = "g1-small"

C:\Users\gagandeep\terra>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:

-/+ destroy and then create replacement

Terraform will perform the following actions:

  # google_compute_instance.vm_instance must be replaced

Plan: 1 to add, 0 to change, 1 to destroy.
```

Working with change

Now let's see an example of **disruptive** change:

Replace the boot disk of your configuration with cos-cloud/cos-stable or any other AMI and re-run terraform plan:

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

Resource actions are indicated with the following symbols:

-/+ destroy and then create replacement

Terraform will perform the following actions:

  # google_compute_instance.vm_instance must be replaced

-/+ resource "google_compute_instance" "vm_instance" {

--

 ~ initialize_params {

          ~ image    = "https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/debian-9-stretch-v20200805"  ->  "cos-cloud/cos-stable"  #
forces replacement

Plan: 1 to add, 0 to change, 1 to destroy.
```

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.

- Run terraform show command to check current required state of infrastructure.
- Modify the Labels (add a label) of a terraform instance from GCP 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.

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.
- Try adding below resource to your configuration file and add a link of same in your Instance network interface:

```
network_interface {
network = google_compute_network.vpc_network.self_link
access_config {
nat_ip = google_compute_address.vm_static_ip.address
 }
 }
resource "google_compute_address" "vm_static_ip" {
 name = "terraform-static-ip"
}
```

Implicit Dependencies

In the previous example, when Terraform reads this configuration, it will:

- Ensure that vm_static_ip is created before vm_instance
- Save the properties of vm_static_ip in the state
- Set nat_ip to the value of the vm_static_ip.address property

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

Implicit dependencies via interpolation expressions are the primary way to inform Terraform about these relationships, and should be used whenever possible.

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.

```
resource "google_storage_bucket" "example_bucket" {
  name     = "<UNIQUE-BUCKET-NAME>"
  location = "US"
  website {
    main_page_suffix = "index.html"
    not_found_page   = "404.html"
  }
}
resource "google_compute_instance" "another_instance" {
depends_on = [google_storage_bucket.example_bucket]
  name         = "terraform-instance-2"
  machine_type = "f1-micro"
  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }
  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
    }
  }
}
```

Explicit Dependencies

- Multiple resource dependencies can also be created:

```
# Create a new instance that uses the bucket
resource "google_compute_instance" "another_instance" {
  # Tells Terraform that this VM instance must be created only after the
  # storage bucket has been created.
  depends_on = [google_storage_bucket.example_bucket1, google_compute_instance.vm_instance]
  name         = "terraform-instance-2"
  machine_type = "f1-micro"

  boot_disk {
    initialize_params {
      image = "cos-cloud/cos-stable"
    }
  }

  network_interface {
    network = google_compute_network.vpc_network.self_link
    access_config {
    }
  }
}
```

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>
```

Note: Terraform determines the order in which things must be destroyed. For e.g. GCP/AWS won't allow a VPC network to be deleted if there are resources still in it, so Terraform waits until the instance is destroyed before destroying the network.

Terraform Advanced

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

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.

```
resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"

  tags = {
    Name = "Techlanders-aws-ec2-instance"
    env = "test"
  }
  provisioner "local-exec" {
    command = "echo The servers IP address is ${self.private_ip} && echo ${self.private_ip} myawsserver >> /etc/hosts"
  }
```

Remote-Exec Provisioners

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

```
resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
provisioner "remote-exec" {
    inline = [
     "touch /tmp/gagandeep",
      "sudo mkdir /root/gagan"
     ]
 connection {
    type     = "ssh"
    user     = "ec2-user"
    insecure = "true"
    private_key = "${file("test1.pem")}"
    host     =  aws_instance.myawsserver.public_ip
  }
 }
 }
```

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"
  }
}
```

Variables

- To become truly shareable and version controlled as well as to avoid hardcoding, we need to parameterize the configurations. Same can be achieved through input variables in Terraform. Variables can be defined in different .tf files and usually we define it in variable.tf or files ending with .tfvars file.

```
variable "project" { }

variable "credentials_file" { }

variable "region" {

  default = "us-central1"

}

variable "zone" {

  default = "us-central1-c"

}
```

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"
 }
```

Variables

- Variables can be assigned via different ways:
  - Via UI
  - Via command line flags:

    terraform plan -var 'project=<PROJECT_ID>’
  - From .tfvars file
  - From environment variables like TF_VAR_name

Variables

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

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Variables

```
resource "aws_instance" "myawsserver1" {
  ami = var.ami["us-east-1"]
  instance_type = var.instance_type
  provider = aws.useast1
  tags = {
    Name = "Techlanders-aws-ec2-instance1"
    Env = "Prod"
  }
}
variable "instance_type" {
  default = "t2.micro"
}

variable "ami" {
  type = "map"
default = {
    us-east-1     = "ami-0c94855ba95c71c99"
    us-east-2     = "ami-0603cbe34fd08cb81"
  }
}
```

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.)

Tfvars files

```
[root@ip-172-31-38-249 third-demo]# cat prod.tfvars
instance_type =  "t2.small"

ami = {
   us-east-1     = "ami-0dba2cb6798deb6d8"
   us-east-2     = "ami-07efac79022b86107 "
 }

[root@ip-172-31-38-249 third-demo]# cat dev.tfvars
instance_type =  "t2.medium"

ami = {
   us-east-1     = "ami-0dba2cb6798deb6d8"
   us-east-2     = "ami-07efac79022b86107 "
 }

[root@ip-172-31-38-249 third-demo]#
```

Executions

```
[root@ip-172-31-38-249 third-demo]# terraform plan
An execution plan has been generated and is shown below.
+ resource "aws_instance" "myawsserver1" {
    + ami                        = "ami-0c94855ba95c71c99"
+ instance_type                  = "t2.micro"

[root@ip-172-31-38-249 third-demo]# terraform plan -var-file="dev.tfvars"
Refreshing Terraform state in-memory prior to plan...
 # aws_instance.myawsserver1 will be created
  + resource "aws_instance" "myawsserver1" {
    + ami                        = "ami-0dba2cb6798deb6d8"
+ instance_type                  = "t2.medium”

[root@ip-172-31-38-249 third-demo]# terraform plan -var "instance_type=t2.nano"
Refreshing Terraform state in-memory prior to plan...
 # aws_instance.myawsserver1 will be created
  + resource "aws_instance" "myawsserver1" {
    + ami                        = "ami-0c94855ba95c71c99"
+ instance_type                  = "t2.nano"
```

Loops

- Terraform offers several different looping constructs, each intended to be used in a slightly different scenario:
  - count parameter: loop over resources.
  - for_each expressions: loop over resources and inline blocks within a resource.
  - for expressions: loop over lists and maps.

Loops - count

- Depending on resource types, it’ll take count values to create number of resources:

```
provider "aws" {
  region = "us-east-2"
 access_key = "AKIAJB2KQBDLH56XQEYA"
  secret_key = "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"
}
resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  count = 2
  tags = {
    Name = "Techlanders-aws-ec2-instance.${count.index}"
    env = "test"
  }
}
output "Private-IP-0" {
 value = aws_instance.myawsserver.0.private_ip
}
output "Private-IP-1" {
 value = aws_instance.myawsserver.1.private_ip
}
```

Loops - count

```
variable "server_names" {
  description = "Create virtual machines with these names"
  type        = list(string)
  default      = ["myvm1", "myvm2"]
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  count = length(var.server_names)
  tags = {
    Name = var.server_names[count.index]
    env = "test"
  }
}

output "Private-IP" {
 value = aws_instance.myawsserver[*].private_ip
}
```

for and for-each

- COUNT have its own limitations. Delete a string from count from previous example and then look at the Terraform behavior. If you remove an item from the middle of the list, Terraform will delete every resource after that item and then recreate those resources again from scratch.
- COUNT can’t be used with-in resource.
- Based on complexity of your playbook, you can use for and for_each loops in your configuration files.
- This is similar to loops in Programming languages.
- https://www.hashicorp.com/blog/hashicorp-terraform-0-12-preview-for-and-for-each/

for-each

```
variable "server_names" {
  description = "Create virtual machines with these names"
  type        = list(string)
  default     = ["vm1", "vm2"]
}

resource "aws_instance" "myawsserver" {
  ami = "ami-0603cbe34fd08cb81"
  instance_type = "t2.micro"
  key_name = "test1"
  for_each = toset(var.server_names)
  tags = {
    Name = each.value
    env = "test"
  }
}

output "Private-IP" {
# As for_each loop is a map, you have to modify the syntax to get the values printed
 value = values(aws_instance.myawsserver)[*].private_ip
}
```

For loop

Terraform’s for expressions allow you to loop over a map using the following syntax:
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]

e.g.

```
output "Private-IP" {
value = {
  for instance in aws_instance.myawsserver:
    instance.id => instance.private_ip
 }
}
```

Conditionals

- Terraform provide conditional values to select one from it:
  - The conditional syntax is the well-known ternary operation:
  - CONDITION ? TRUEVAL : FALSEVAL

```
resource "aws_instance" "web" {
  subnet = "${var.env == "production" ? var.prod_subnet : var.dev_subnet}"
}
```

Statefiles

- 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 lockfile for security

- Terraform acquires a state lock to protect the state from being written by multiple users at the same time.
- Just run terraform apply or destroy and don't provide any input on its confirmation command. Open a new terminal and look for .terraform.tfstate.lock.info file being created in the directory.

```
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>
```

- State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue.
- You can disable state locking for most commands with the -lock flag but it is not recommended.
- You can unlock terraform with terraform force-unlock LOCK_ID command.

State files

- 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.
- Terraform uses this local state to create plans and make changes to your infrastructure. Prior to any operation, Terraform does a refresh to update the state with the real infrastructure.
- 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.
- State snapshots are stored in JSON format
- What will happen if multiple people from a team are working on the requirement?
- How code will me managed and how state file will be maintained?

Remote State

- When working with Terraform in a team, use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.

- With remote state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Alibaba Cloud OSS, and more.

- Here are some of the benefits of backends:
  - **Working in a team:** Backends can store their state remotely and protect that state with locks to prevent corruption. Some backends such as Terraform Cloud 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.

Remote State- GCS

- Modify your configuration file to handle state remotely:

```
provider "google" {
version = "3.5.0"
credentials = file("accenture-286519-79f7f889142f.json")
project = "accenture-286519"
  region  = "us-central1"
}
terraform {
  backend "gcs" {
    bucket  = "gagantechlanders1"
    prefix  = "terraform/state"
  }
}
resource "google_compute_network" "vpc_network" {
  name = "terraform-network4"
}
```

Remote State - AWS

```
terraform {
  backend "s3" {
    bucket  = "techlanders-statefile"
    key  = "terraform/state"
    region = "us-east-2"
   access_key = "AKIAJLH56XQEYA"
   secret_key = "rNNWOXCB10Zr2OVuPI3iayxXXStPs"
  }
}
```

```
[root@ip-172-31-38-249 loops]# terraform init

Initializing the backend...

Successfully configured the backend "s3"! Terraform will automatically

use this backend unless the backend configuration changes.
```

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
```

Terraform Modules

```
module "consul" {
  source  = "hashicorp/consul/aws"
  version = "0.0.5"
  servers = 3
}
```

Importing existing resources

- You can import existing resources – which are not created using terraform command, into terraform state using terraform import command.
- The current implementation of Terraform import can only import resources into the state. It does not generate configuration. A future version of Terraform will also generate configuration.
- Because of this, prior to running terraform import it is necessary to write a resource configuration block for the resource manually, to which the imported object will be attached.
- This command will not modify your infrastructure, but it will make network requests to inspect parts of your infrastructure relevant to the resource being imported.

Importing existing resources

```
[root@techlanders]# cat resource.tf
resource "aws_instance" "gagan-ec2" {
  ami           = "ami-0c94855ba95c71c99"
  instance_type = "t2.micro"
  availability_zone = "us-east-1e"
}
[root@techlanders]#

#use terraform import command to import state in terraform statefile
[root@techlanders]# terraform import aws_instance.gagan-ec2 i-073cd0c68788f5c57
[root@techlanders]# terraform show
[root@techlanders]# terraform plan
[root@techlanders]# terraform apply
```

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

```
[root@ip-172-31-38-249 aws]# terraform taint aws_instance.myawsserver
Resource instance aws_instance.myawsserver has been marked as tainted.
[root@ip-172-31-38-249 aws]# terraform plan
Resource actions are indicated with the following symbols:
-/+ destroy and then create replacement
Terraform will perform the following actions:
  # aws_instance.myawsserver is tainted, so must be replaced
-/+ resource "aws_instance" "myawsserver"
```

Note: This command on its own will not modify infrastructure. For same you'll have to run terraform apply.

- You can use untaint command to untaint a node.

```
[root@ip-172-31-38-249 aws]# terraform untaint aws_instance.myawsserver
Resource instance aws_instance.myawsserver has been successfully untainted.
[root@ip-172-31-38-249 aws]#
```

Workspaces

- Workspace is to create multiple isolated environments in same directory.
- Terraform starts with a single workspace named "default" and same can't be deleted.
- Named workspaces allow conveniently switching between multiple instances of a single configuration within its single backend.
- A common use for multiple workspaces is to create a parallel, distinct copy of a set of infrastructure in order to test a set of changes before modifying the main production infrastructure. For example, a developer working on a complex set of infrastructure changes might create a new temporary workspace in order to freely experiment with changes without affecting the default workspace.
- It'll create terraform.tfstate.d directory with internal workspace-name subdirectories to handle state files.

Workspaces

- terraform workspace list
- terraform workspace new {new-workspace-name}
- terraform workspace show
- terraform workspace select {workspace-name}
- terraform workspace delete {workspace-name}

Best Practices

Best Practices

- Terraform recognizes files ending in .tf or .tf.json as configuration files and will load them when it runs. When run, Terraform loads all configuration files from the current directory. So it's a good idea to organize your configurations into separate directories based on your needs (e.g. departments, production vs development, etc).
- Successful execution of terraform plan doesn’t mean actual implementation will be always successful. It may fail due to provider parameters issue. For example in case you provide an image name which doesn’t exists, terraform will take it and assume it’ll be available, and plan will get successful. So a real run can only provide you better assurance.
- Even a Real run sometime will not be helpful if your backend environment is changing. For example, a perfect execution of infrastructure deployment in your aws environment doesn't mean it’ll work in client environment too, if you have hardcoded the things. For example, in case you hardcode keypair or network in your configuration file.
- Sometimes even configuration parameters become hardcode and creates problem. For example a bucket name must be globally unique in GCP. If you have used that name already somewhere in your test/dev account, same can not be used in prod or client accounts – which is configured under bucket name section under terraform configuration file. So consider this factor too.

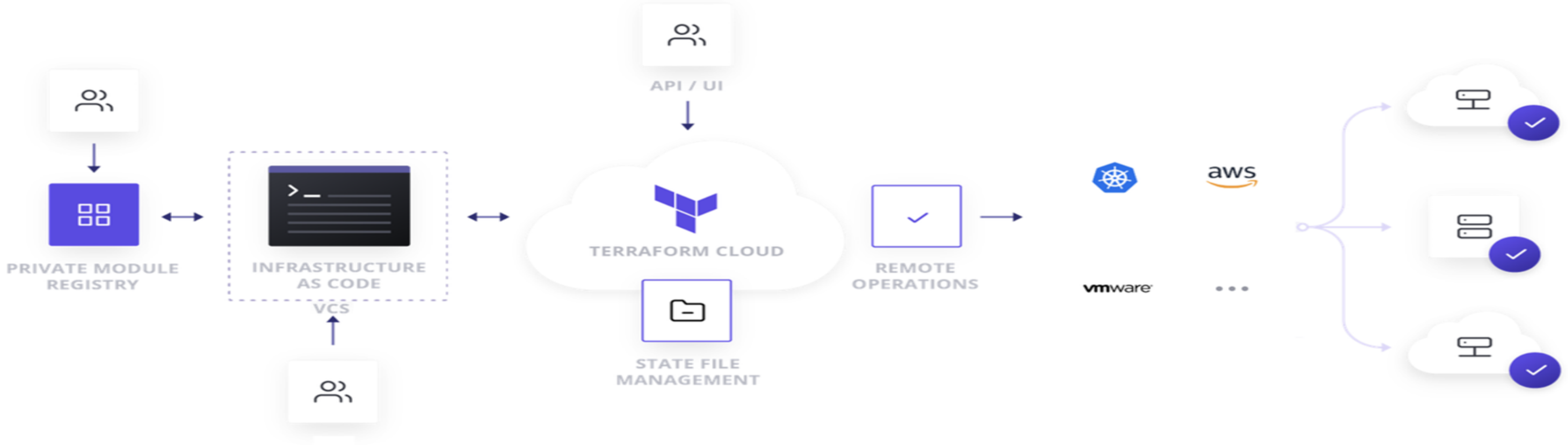
Best Practices

- During working inside a team, always keep your configuration files on Github or similar VCS.
- Keep your remote state on S3/GCS (with versioning enabled) for HA and collaborative working.
- Follow DRY Principle. The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".
- Have thumb rule- **Let the expert do its job**. Use Terraform for Infra provisioning only, not for Configuration management. Same rules applied to CM tools (for not to use them for Infra provisioning).
- Do the automation wherever possible.
- Hardcoding will hinder the real automation needs. So make sure to put the things in variables wherever possible.

Terraform Cloud

Terraform Cloud

Terraform Cloud is an application that helps teams use Terraform together. It manages Terraform runs in a consistent and reliable environment and includes easy access to shared state and secret data, access controls for approving changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations, and more.



Terraform Cloud

It is a platform that performs Terraform runs to provision infrastructure, either on demand or in response to various events

Terraform Cloud offers a **team-oriented remote Terraform workflow**, designed to be comfortable for existing Terraform users and easily learned by new users. The foundations of this workflow are remote Terraform execution, a **workspace-based organizational model**, **version control integration**, **command-line integration, remote state management** with cross-workspace data sharing, and a **private Terraform module registry.**

Terraform Cloud runs Terraform on **disposable virtual machines in its own cloud infrastructure**. **Remote Terraform execution** is sometimes referred to as "remote operations."

Terraform Cloud

- Terraform cloud is a GUI based Cloud SaaS solution. It is offered as a multi-tenant SaaS platform and is designed to suit the needs of smaller teams and organizations.
- Benefits of Terraform Cloud:
    - It manages Terraform runs in a consistent and reliable environment.
    - Best for bigger teams, as it provides secure and easy access to shared state and secret data.
    - It offers Remote State Management, Data Sharing, Run Triggers, and Private registry for Terraform modules.
    - Role Based Access Controls (RBAC) for approving changes to infrastructure.
    - Version Control Integration with Major VCS providers like Github, Gitlab, Bitbucket, Azure DevOps
    - Full APIs support for all operations to integrate this with other tools and environments.

Terraform Cloud

- Notifications can be configured with services which support webhooks
- You can run the configuration from existing environment or from terraform cloud-based server.
- **Sentinel Policies:** Terraform Cloud embeds the Sentinel policy-as-code framework, which lets you define and enforce granular policies for how your organization provisions infrastructure. You can limit the size of compute VMs, confine major updates to defined maintenance windows, and much more. Policies can act as firm requirements, advisory warnings, or soft requirements that can be bypassed with explicit approval from your compliance team.
- **Cost Estimation**: Before making changes to infrastructure in the major cloud providers, Terraform Cloud can display an estimate of its total cost, as well as any change in cost caused by the proposed updates.

Terraform Enterprise

- Terraform Enterprise is a self-hosted distribution of Terraform Cloud Application.
- Provides additional security as everything is on-prem.
- It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on.

Lab: Cloud

1) Open webpage https://app.terraform.io and create an account there

2) Create an Organization by providing name and Email address

3) Create a workspace

Select workflow with VCS (another options like CLI and API driven can also be used)

Integrate your version control system (I have selected Github for this example)

To integrate Github, open link https://github.com/settings/applications/new and provide information been provided by terraform registration page

Setup Oauth authentication as guided by setting up the details

Authorise Terraform cloud to have admin access on your repositories

4) Skip the ssh-keypair as same is not required to setup. SSH-keypair is basically to connect to git repos via ssh, which sometimes required when you have private submodules.

5) Select the repositories where you have your terraform code placed. Don't forget to select the working directory(under advanced section) where you want to work on, especially when you have multiple terraform folders inside the repo. Terraform working directory can be changed lateron from Settings -> General tab.

Lab: Cloud

6) Click on create workspace and finish the creation.

7) Add Terraform Environment Variable and set AWS AK/SK(AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY). Don't forget to select sensitive option on the right side, especially for Secret Access Key.

8) Click on Queue Plan and provide a reason(just any description and better to have though optional) for queuing same.

9) Check the planning logs

10) Apply the changes and look at the output segment of apply logs

11) Cross-check the state under state section post your apply is completed. You can verify your state file outcome, run state and version of the source code etc. You can even download the tfstate file, by clicking on download button.

12) Below add-features can be enabled/disabled/modified:

a) Auto-approve, Remote/Local execution, Terraform version, working directory :   Under settings -> General Tab.

b) Manually locking a project. By default Lock is auto-applied during terraform apply. : Under settings -> Locking

c) Email/Webhooks/Slack Notifications - Under Settings -> Notifications

d) Triggering the workspace based on run of another workspace : settings -> Run Triggers

e) Source Code Integration and SSH key setup, can be done via : Settings -> SSH/Version control

Lab: Cloud

Automated pipelining:

13) Change the configuration to auto-approve and do the modification on github under your configuration change and cross-check the terraform apply.

Terraform Enterprise

- Terraform Enterprise is a self-hosted distribution of Terraform Cloud Application.
- Provides additional security as everything is on-prem.
- It offers enterprises a private instance of the Terraform Cloud application, with no resource limits and with additional enterprise-grade architectural features like audit logging and SAML single sign-on.

HashiCorp

Packer

Packer

Hashicorp packer is used to automates the creation of any type of machine image.

It embraces modern configuration management by encouraging you to use automated scripts to install and configure the software within your Packer-made images.

Similar to Terraform, Packer also uses the Hashicorp Configuration Language – HCL, for its configuration file.

Packer can create multiple images for multiple platforms in parallel, all configured from a single template.

Builders in Packer are responsible for creating machines and generating images from them for various platforms. For example, there are separate builders for EC2, VMware, VirtualBox, etc. Packer comes with many builders by default.

Packer Installation

```
sudo yum install -y yum-utils

sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo

sudo yum -y install packer

Packer --version
```

NOTE: There is an existing package management tool with name packer and you may find same in /usr/sbin/ directory. Remove/rename that packer to use this as your default command. Else it may redirect you to use that packer which is package manager.

```
[root@techlanders /]# packer --version

1.6.2

[root@techlanders /]#
```

Templates

Templates are JSON files that configure the various components of Packer in order to create one or more machine images. Templates are portable, static, and readable and writable by both humans and computers.

```
{
  "builders": [
    {
      "type": "amazon-ebs",
      "access_key": "{{user `aws_access_key`}}",
      "secret_key": "{{user `aws_secret_key`}}",
      "region": "us-east-1",
      "source_ami": "ami-0c94855ba95c71c99",
      "instance_type": "t2.micro",
      "ssh_username": "ubuntu",
      "ami_name": "my-packer-ami {{timestamp}}"
    }
  ]
}
```

Templates Structure

Templates are JSON files that configure the various components of Packer in order to create one or more machine images. Templates are portable, static, and readable and writable by both humans and computers.

- **builders** (required) is an array of one or more objects that defines the builders that will be used to create machine images for this template, and configures each of those builders.
- **min_packer_version** (optional) is a string that has a minimum Packer version that is required to parse the template. This can be used to ensure that proper versions of Packer are used with the template.
- **provisioners** (optional) is an array of one or more objects that defines the provisioners that will be used to install and configure software for the machines created by each of the builders.
- **Post-processors** (optional) run after the image is built by the builder and provisioned by the provisioner(s). Post-processors are optional, and they can be used to upload artifacts, re-package, or more.
- **variables** (optional) is an object of one or more key/value strings that defines user variables contained in the template.

JSON doesn't support comments and Packer reports unknown keys as validation errors. If you'd like to comment your template, you can prefix a root level key with an underscore.

Templates

Templates are JSON files that configure the various components of Packer in order to create one or more machine images. Templates are portable, static, and readable and writable by both humans and computers.

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [
    {
      "type": "amazon-ebs",
      "access_key": "{{user `aws_access_key`}}",
      "secret_key": "{{user `aws_secret_key`}}",
      "region": "us-east-1",
      "source_ami_filter": {
        "filters": {
          "virtualization-type": "hvm",
          "name": "ubuntu/images/*ubuntu-xenial-16.04-amd64-server-*",
          "root-device-type": "ebs"
        },
        "owners": ["099720109477"],
        "most_recent": true
      },
      "instance_type": "t2.micro",
      "ssh_username": "ubuntu",
      "ami_name": "packer-example {{timestamp}}"
    }
  ]
}
```

Templates

```
[root@techlanders /]# packer validate -syntax-only image1.json
Syntax-only check passed. Everything looks okay.
[root@techlanders /]#  packer validate  image1.json
[root@techlanders /]#
[root@techlanders /]# packer inspect image1.json
Packer Inspect: JSON mode
Optional variables and their defaults:
  aws_access_key = AKIAJQIXWFGW3OAYHHRQ
  aws_secret_key = rjsqh/yaTDKh6JoIcADCcNGoxIV6L2cHcn2f06vC
Builders:
  amazon-ebs
Provisioners:
  <No provisioners>
Note: If your build names contain user variables or template
functions such as 'timestamp', these are processed at build time,
and therefore only show in their raw form here.
[root@techlanders /]#
```

Templates

```
[root@techlanders /]# packer build image1.json
amazon-ebs: output will be in this color.

==> amazon-ebs: Prevalidating any provided VPC information
==> amazon-ebs: Prevalidating AMI Name: packer-example 1600481261
    amazon-ebs: Found Image ID: ami-0f82752aa17ff8f5d
==> amazon-ebs: Creating temporary keypair: packer_5f6567ed-df5d-2613-c37f-5f48bb16870f
==> amazon-ebs: Creating temporary security group for this instance: packer_5f6567ee-7852-926a-e7a8-064298c7ab91
==> amazon-ebs: Authorizing access to port 22 from [0.0.0.0/0] in the temporary security groups...
==> amazon-ebs: Launching a source AWS instance...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Stopping the source instance...
    amazon-ebs: Stopping instance
==> amazon-ebs: Creating AMI packer-example 1600481261 from instance i-0bb03c323eb86c588
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...
Build 'amazon-ebs' finished after 2 minutes 59 seconds.
==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:
us-east-1: ami-0af89fb38a1b1859d

[root@techlanders /]#
```

Builds

HCL2 support for Packer is still in Beta. So currently json is supported, soon hcl will be used for this.

Packer build command takes one argument. When a directory is passed, all files in the folder with a name ending with .pkr.hcl or .pkr.json.

The build block defines what builders are started, how to provision them and if necessary, what to do with their artifacts using post-process.

hcl2_upgrade

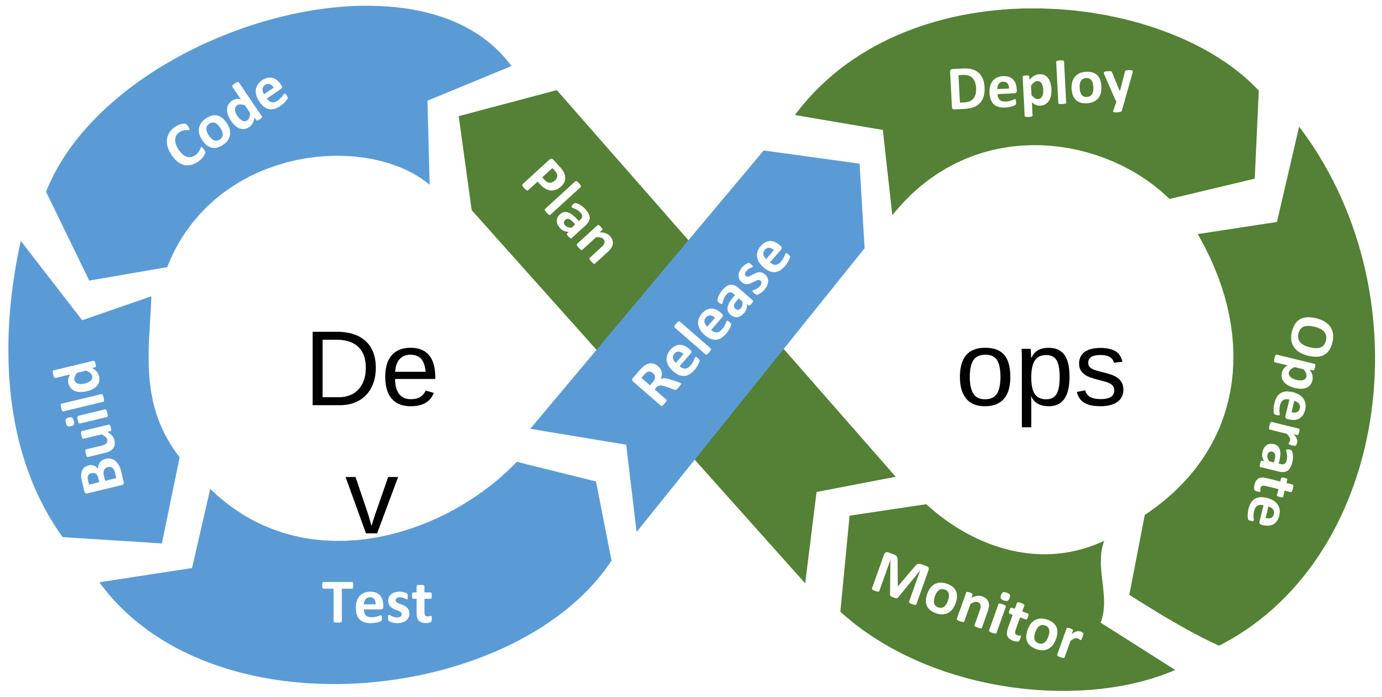
The packer hcl2_upgrade Packer command is used to transpile a JSON configuration template to it's formatted HCL2 counterpart. The command will return a zero exit status on success, and a non-zero exit status on failure.

```
$ packer hcl2_upgrade my-template.json

Successfully created my-template.json.pkr.hcl
```

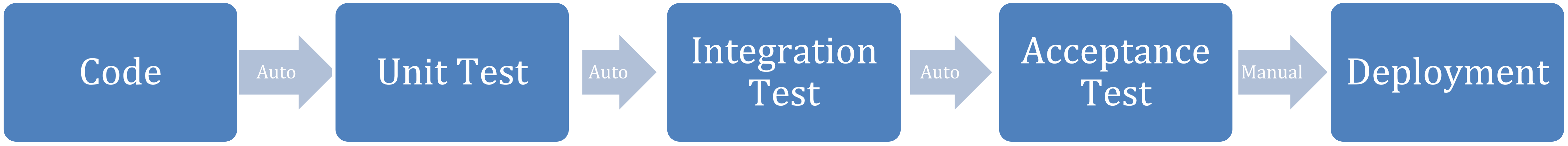
Terraform in DevOps

Terraform in DevOps

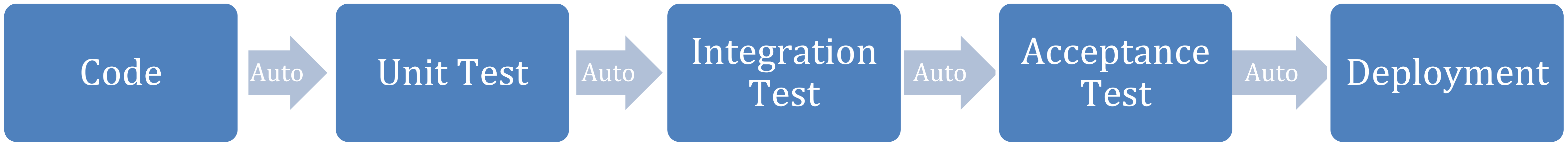


Terraform in DevOps

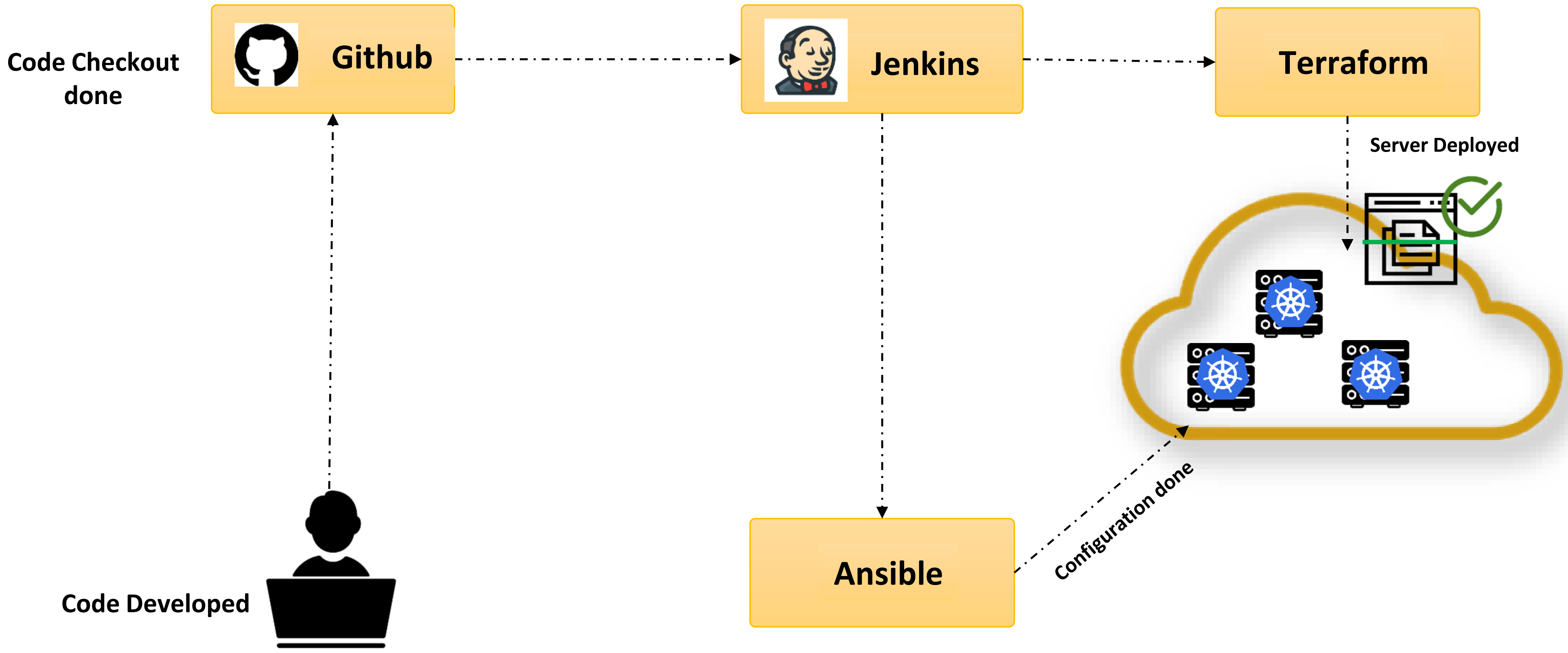
Continuous Integration



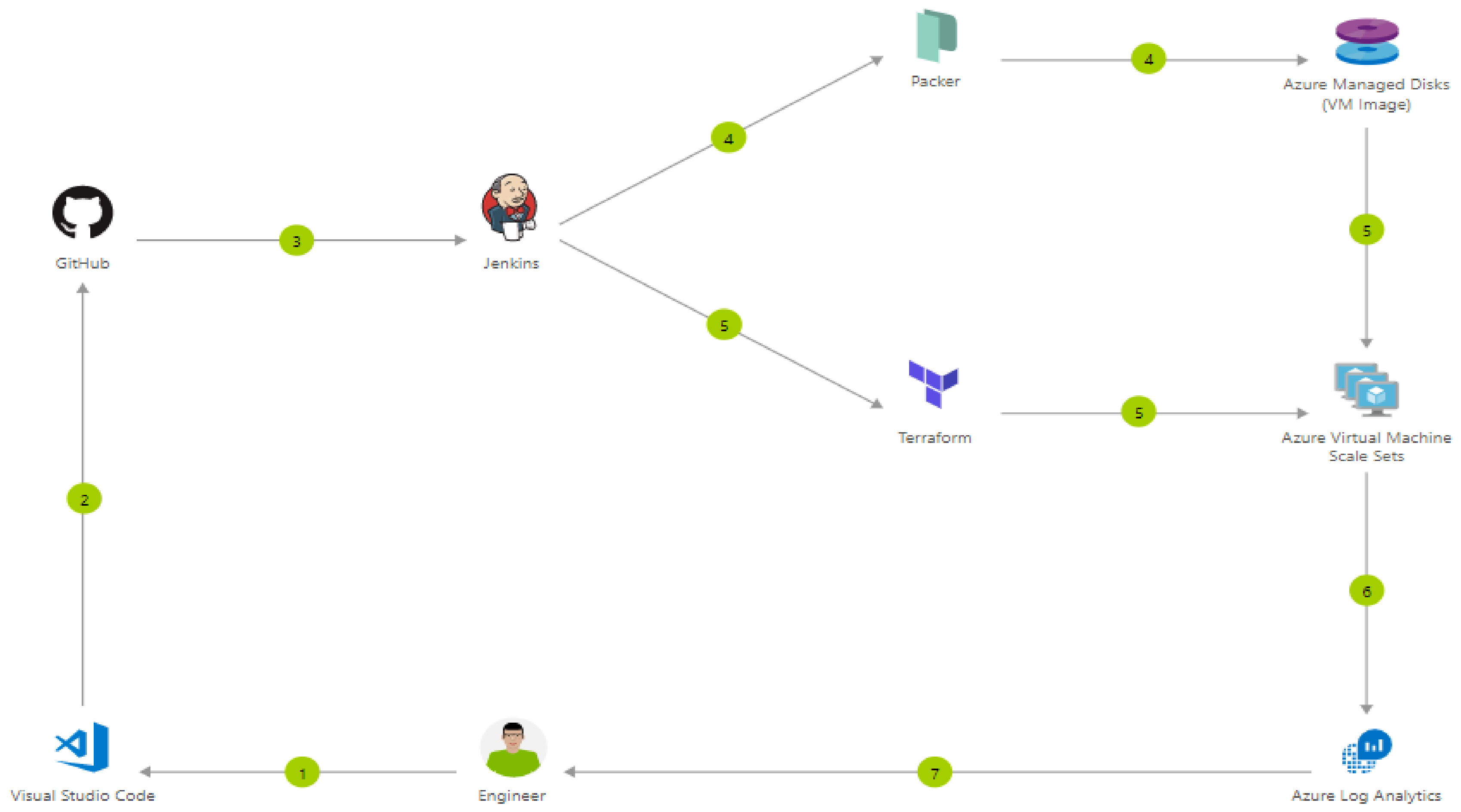
Continuous Deployment



Terraform in DevOps



DevOps IaC



Packer
4
Azure Managed Disks (VM Image)
4
GitHub
3
Jenkins
5
5
Terraform
5
Azure Virtual Machine Scale Sets
2
6
Visual Studio Code
1
Engineer
7
Azure Log Analytics

Terraform in DevOps

Install git client
yum install git

Install Jenkins client
sudo yum update -y
sudo yum install java-1.8.0-openjdk.x86_64 -y
java –version
Set JAVA_HOME and JRE_HOME
export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk
export JRE_HOME=/usr/lib/jvm/jre
yum install wget -y

sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo
sudo rpm --import http://pkg.jenkins-ci.org/redhat-stable/jenkins-ci.org.key
sudo yum install jenkins

sudo systemctl start jenkins.service
sudo systemctl enable jenkins.service

Run Jenkins with root

Lab: Ansible in DevOps

Create a new project in Jenkins:

1) Add Plugins – Git
2) Create repository on GitHub and add your playbooks on same
3) Add path of your GitHub in newly created project and set polling for every minute
4) Add build step to run the terraform apply statement
5) Build it for the first time
6) Do the changes in GitHub repo and observe the auto execution of next build on it own.

Questions & Answers

THANK
YOU