Atgen

Terraform

Enablement Workshop

Agenda

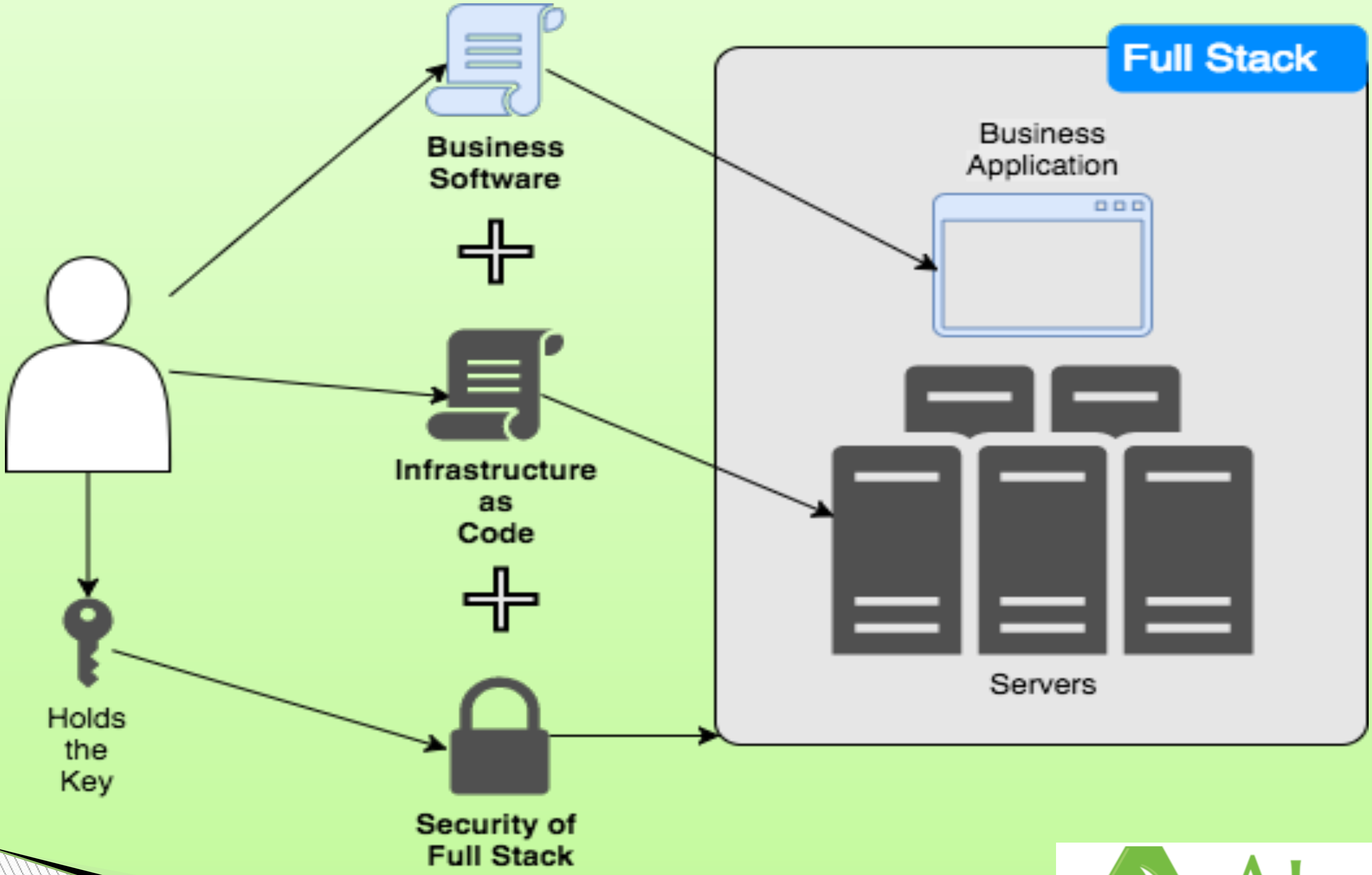
- What is IaC?
- DevOps – Terraform
- Terraform Setup
- Terraform Configurations
- Terraform Conditional Statements
- Terraform – Templates/Modules
- Terraform - Utility Resources

Session: 1

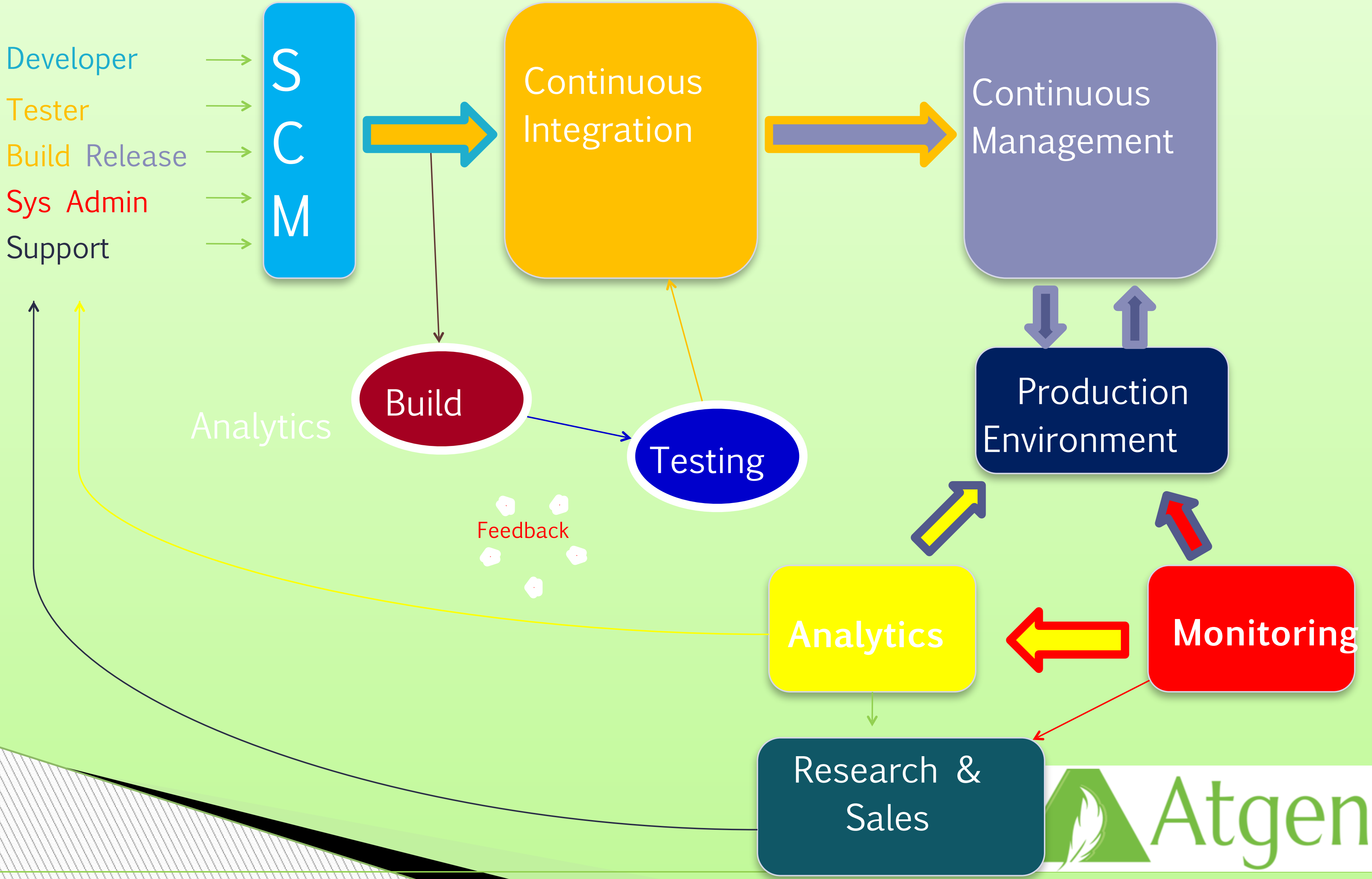
Infrastructure as Code

Atgen

What is Business Service?



What is DevOps?



Developer

Tester

Build Release

Sys Admin

Support

SCM

Continuous Integration

Continuous Management

Build

Testing

Analytics

Feedback

Production Environment

Analytics

Monitoring

Research & Sales

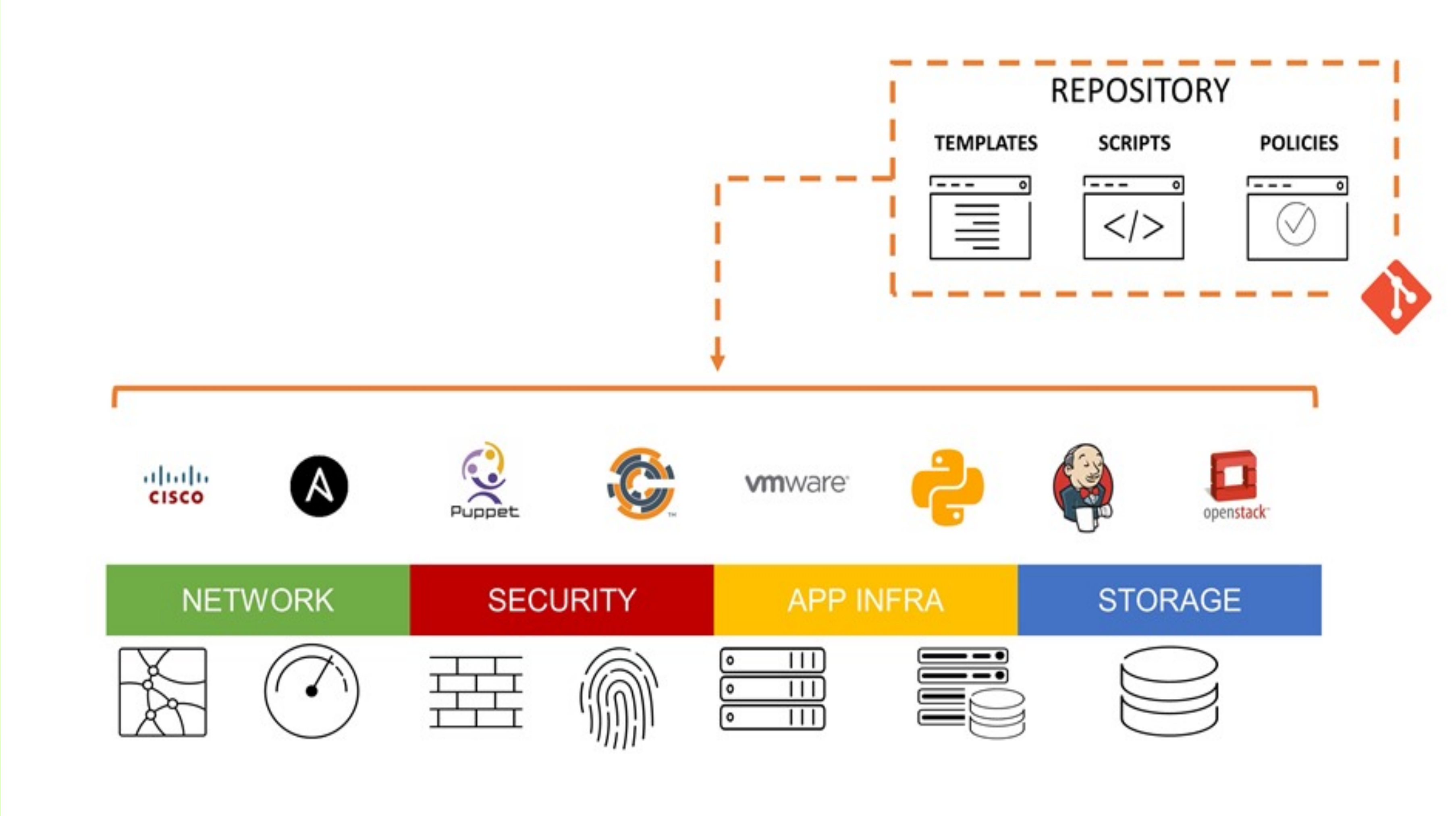
What is IaC?

- Infrastructure as code, also referred to as IaC, is a type of IT setup wherein developers or operations teams automatically manage and provision the technology stack for an application through software, rather than using a manual process to configure discrete hardware devices and operating systems.
- Infrastructure as code is sometimes referred to as programmable or software-defined infrastructure.
- The concept of infrastructure as code is similar to programming scripts, which are used to automate IT processes. However, scripts are primarily used to automate a series of static steps that must be repeated numerous times across multiple servers.

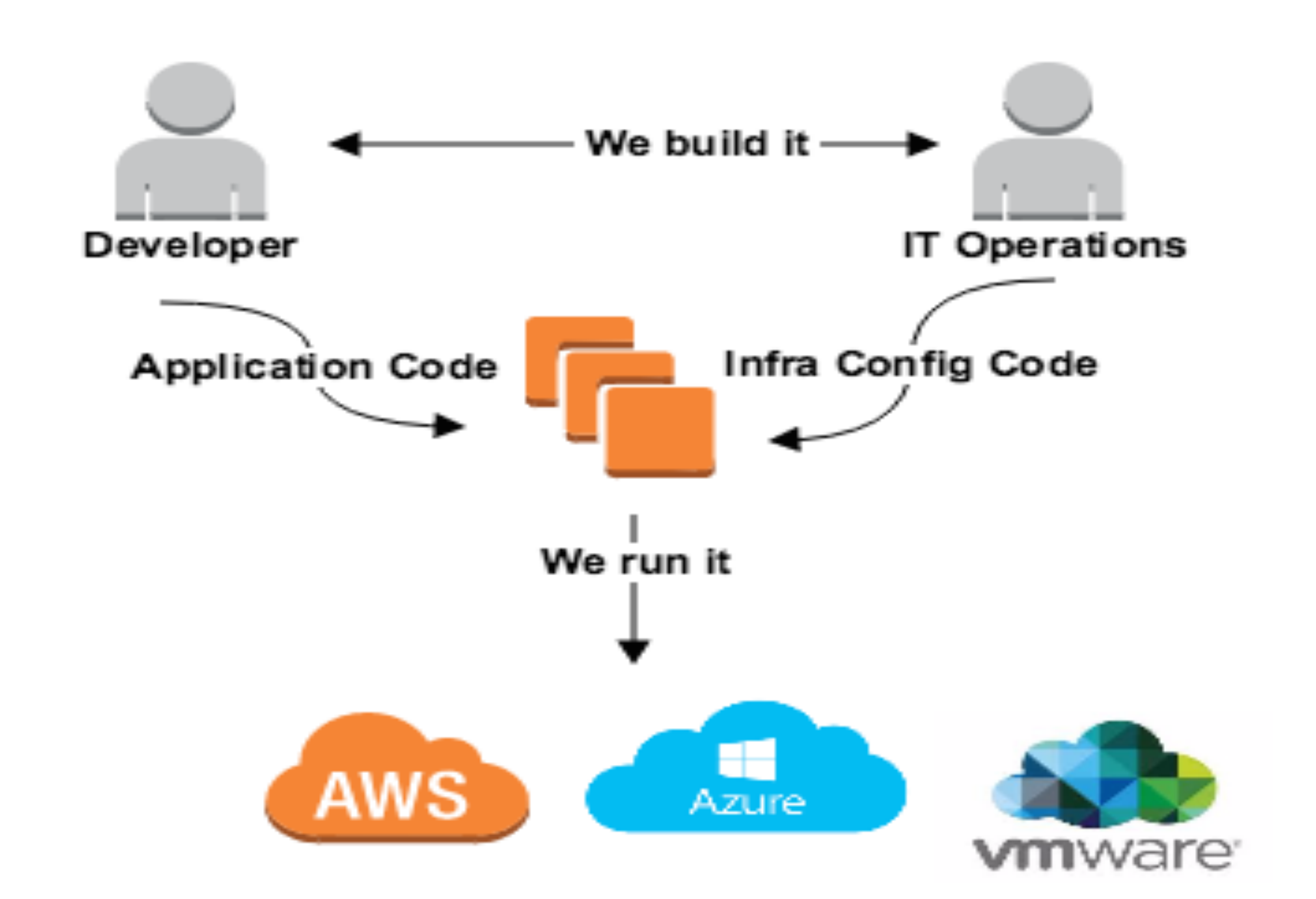
What is IaC?

- Infrastructure as code uses higher-level or descriptive language to code more versatile and adaptive provisioning and deployment processes. For example, infrastructure-as-code capabilities included with Terraform, an IT management and configuration tool, can install WebServer, verify that WebServer is running properly, create a user account and password.
- The code-based infrastructure automation process closely resembles software design practices in which developers carefully control code versions, test iterations, and limit deployment until the software is proven and approved for production.

What is IaC?

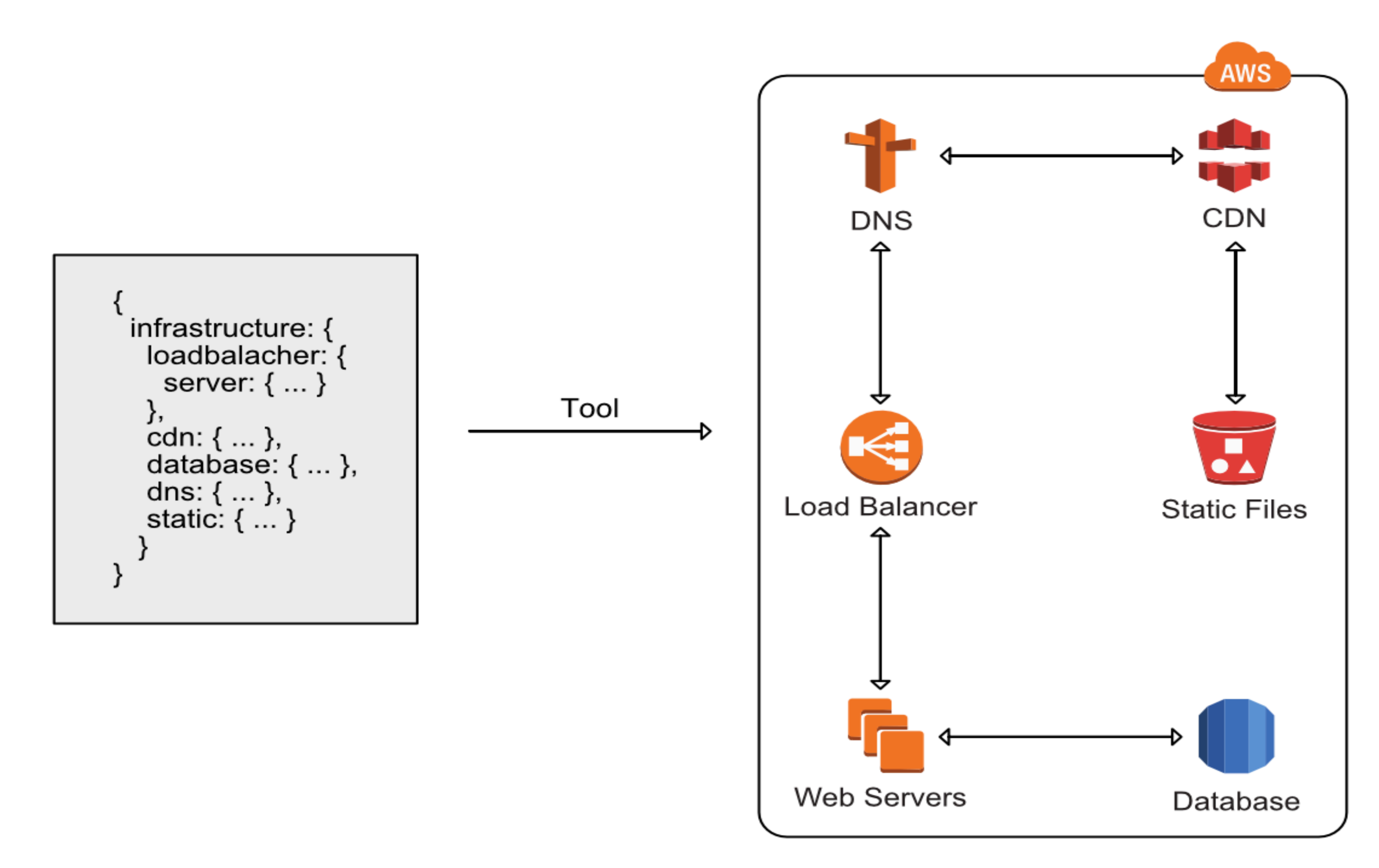

REPOSITORY
TEMPLATES
SCRIPTS
POLICIES
cisco
Puppet
vmware
openstack
NETWORK
SECURITY
APP INFRA
STORAGE

Where we stand?



Atgen

IaC – Example



```
{
  infrastructure: {
    loadbalacher: {
      server: { ... }
    },
    cdn: { ... },
    database: { ... },
    dns: { ... },
    static: { ... }
  }
}
```

Benefits of IaC

- Software developers can use code to provision and deploy servers and applications, rather than rely on system administrators in a DevOps environment.
- With the infrastructure setup written as code, it can go through the same version control, automated testing, and other steps of a continuous integration and continuous delivery(CI/CD) pipeline that developers use for application code.
- Because the OS and hardware infrastructure is provisioned automatically and the application is encapsulated atop it, these technologies prove complementary for diverse deployment targets, such as test, staging and production.

Benefits of IaC

- Despite its benefits, infrastructure as code poses potential disadvantages. It requires additional tools, such as a configuration management system, that could introduce learning curves and room for error.
- If administrators change server configurations outside of the set infrastructure-as-code template, there is potential for configuration drift. It's important to fully integrate infrastructure as code into systems administration, IT operations and DevOps practices with well-documented policies and procedures.

Session: 2

DevOps - Terraform

What is Terraform?

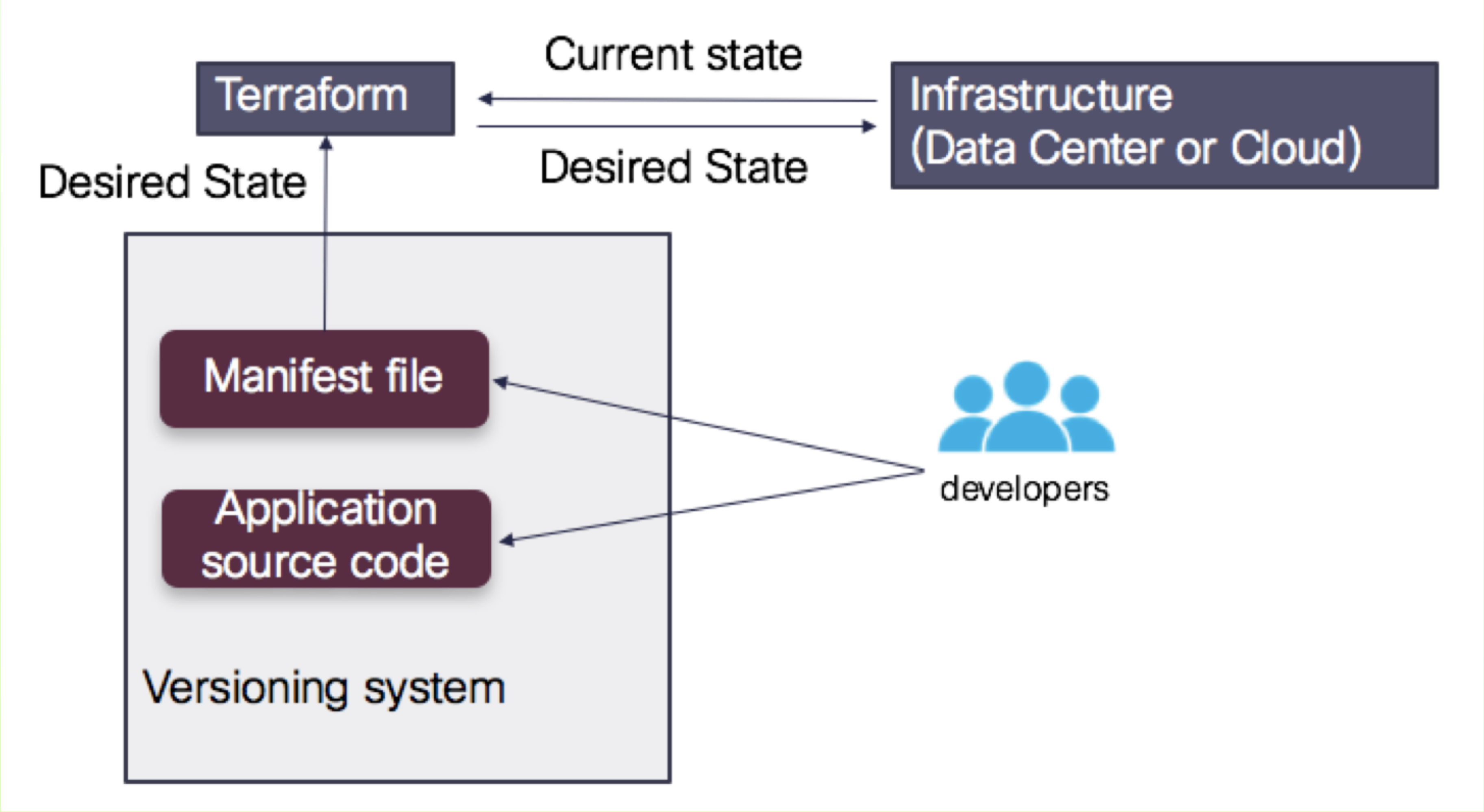
- Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
- Configuration files describe to Terraform the components needed to run a single application or your entire datacenter.
- Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure.
- As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.
- The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

Atgen

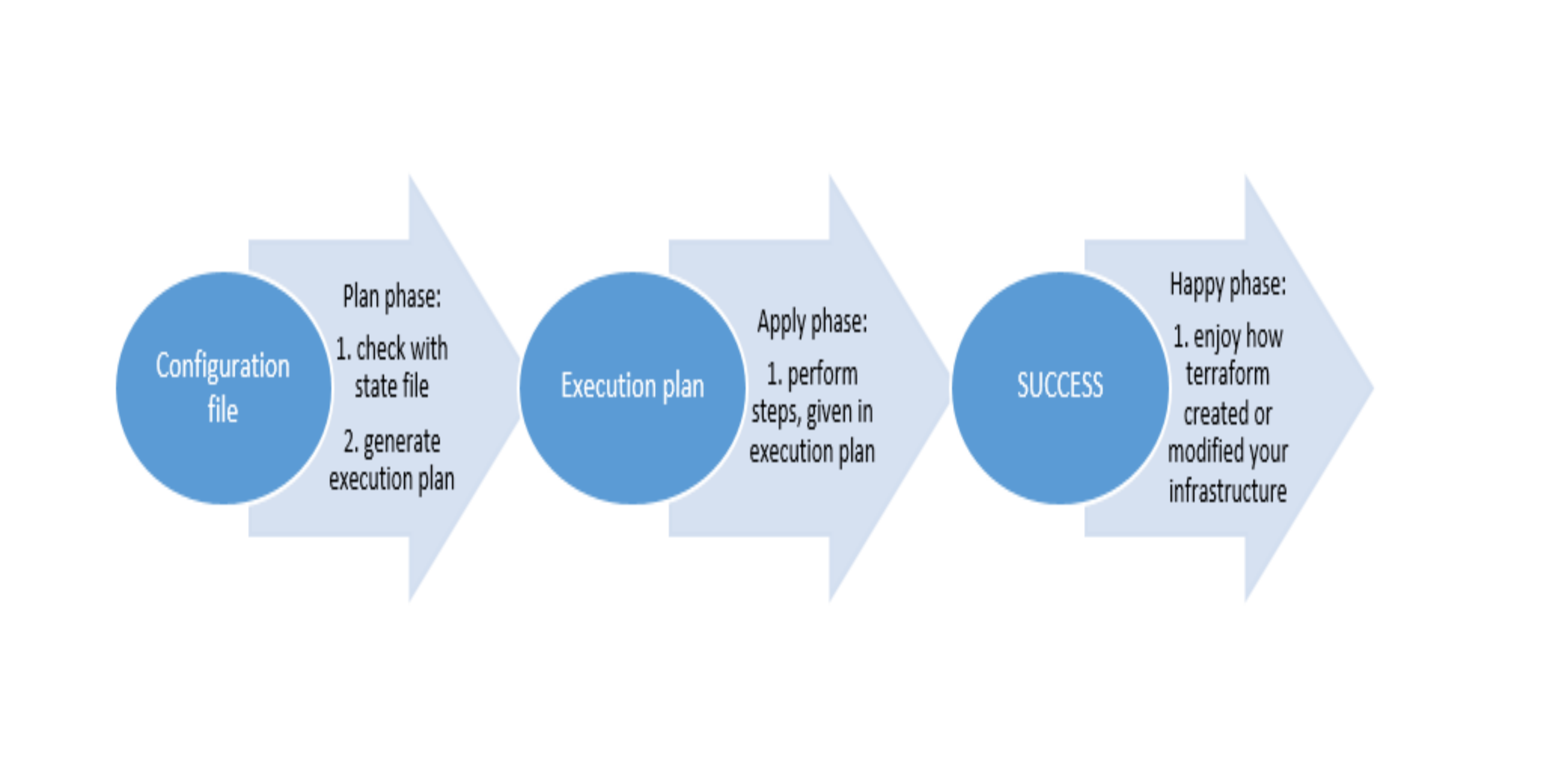
Features of Terraform

- Infrastructure as Code(IaC)
- Execution Plans
- Resource Graph
- Change Automation
- Collaboration

How Terraform works?



How Terraform works?



Configuration file

Plan phase:

1. check with state file

2. generate execution plan

Execution plan

Apply phase:

1. perform steps, given in execution plan

SUCCESS

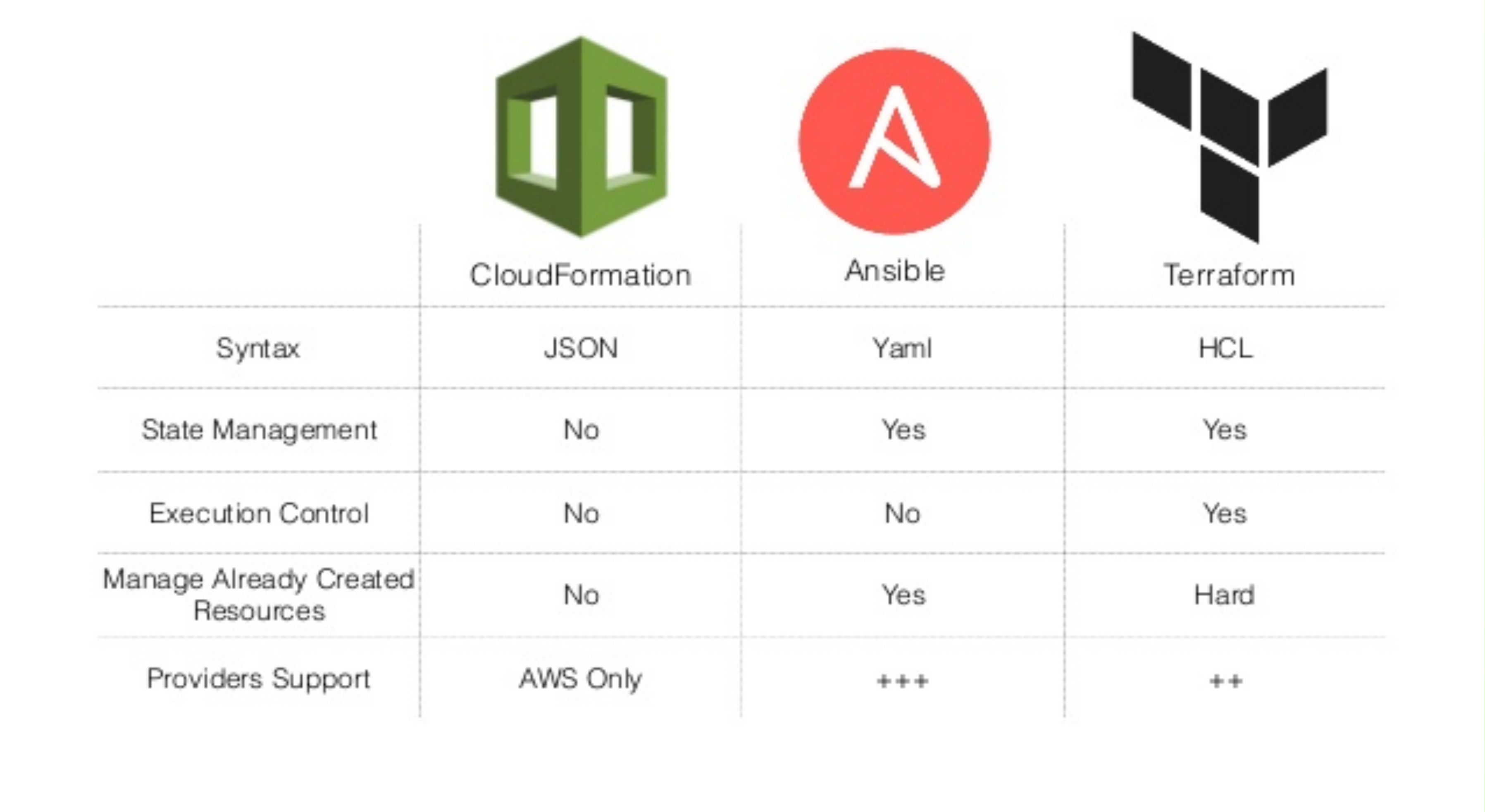
Happy phase:

1. enjoy how terraform created or modified your infrastructure

Terraform vs Puppet,Chef

- Configuration management tools install and manage software on a machine that already exists.
- Terraform is not a configuration management tool, and it allows existing tooling to focus on their strengths: bootstrapping and initializing resources.
- Using provisioners, Terraform enables any configuration management tool to be used to setup a resource once it has been created.
- Terraform focuses on the higher-level abstraction of the datacenter and associated services, without sacrificing the ability to use configuration management tools to do what they do best. It also embraces the same codification that is responsible for the success of those tools, making entire infrastructure deployments easy and reliable.

Terraform vs Ansible,CF



	CloudFormation	Ansible	Terraform
Syntax	JSON	Yaml	HCL
State Management	No	Yes	Yes
Execution Control	No	No	Yes
Manage Already Created Resources	No	Yes	Hard
Providers Support	AWS Only	+++	++

Session: 3

Classroom Environment

Terraform – Lab Setup

- Terraform will be installed for Lab Environment.
- Steps:

```
 yum install -y yum-utils
 yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
yum -y install terraform
```

Session: 4

Terraform - Configuration

Terraform – Configuration

- Terraform uses text files to describe infrastructure and to set variables. These text files are called Terraform *configurations* and end in .tf.
- The format of the configuration files are able to be in two formats: Terraform format and JSON.
- The Terraform format is more human-readable, supports comments, and is the generally recommended format for most Terraform files.
- The JSON format is meant for machines to create, modify, and update.

Load Order & Semantics

- When invoking any command that loads the Terraform configuration, Terraform loads all configuration files within the directory specified in alphabetical order.
- The files loaded must end in either .tf or .tf.json to specify the format that is in use. Otherwise, the files are ignored.
- Override files are the exception, as they're loaded after all non-override files, in alphabetical order.
- The configuration within the loaded files are appended to each other.
- The order of variables, resources, etc. defined within the configuration doesn't matter.

Configuration Syntax

- The syntax of Terraform configurations is called HashiCorp Configuration Language (HCL).
- It is meant to strike a balance between human readable and editable as well as being machine-friendly.
- For machine-friendliness, Terraform can also read JSON configurations.
- For general Terraform configurations, however, we recommend using the HCL Terraform syntax.

Configuration Syntax

```
# An AMI
variable "ami" {
  description = "the AMI to use"
}
/* A multi
   line comment. */
resource "aws_instance" "web" {
  ami                = "${var.ami}"
  count              = 2
}
```

Providers

- Terraform is used to create, manage, and update infrastructure resources such as physical machines, VMs, network switches, containers, and more.
- Almost any infrastructure type can be represented as a resource in Terraform.
- A provider is responsible for understanding API interactions and exposing resources.
- Providers generally are an IaaS (e.g. AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Enterprise, DNSimple, CloudFlare).

Providers

- All the terraform providers can be found at:

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

Provider Configurations

- Providers are responsible in Terraform for managing the lifecycle of a resource: create, read, update, delete.
- Most providers require some sort of configuration to provide authentication information, endpoint URLs, etc.
- By default, resources are matched with provider configurations by matching the start of the resource name. For example, a resource of type vsphere_virtual_machine is associated with a provider called vsphere.

Provider Configurations - Example

- A provider configuration looks like the following:

```
provider "aws" {
    access_key = "foo"
    secret_key = "bar"
    region     = "us-east-1"
}
```

Provider Initialisations

- Each time a new provider is added to configuration, it's necessary to initialise that provider before use.
- Initialisation downloads and installs the provider's plugin and prepares it to be used.
- Provider initialisation is one of the actions of terraform init. Running this command will download and initialise any providers that are not already initialised.
- Providers downloaded by terraform init are only installed for the current working directory.
- Note that terraform init cannot automatically download providers that are not distributed by HashiCorp.

Multiple Provider Instances

- You can define multiple configurations for the same provider in order to support multiple regions, multiple hosts, etc.

```
# The default provider configuration
    provider "aws" {
        # ...
    }
# Additional provider configuration for west coast region
    provider "aws" {
        alias  = "west"
        region = "us-west-2"
    }
```

Multiple Provider Instances

- Using provider in resource:

```
resource "aws_instance" "foo" {
    provider = "aws.west"
     # ...
}
```

Resource Configurations

- The most important thing you'll configure with Terraform are resources.
- Resources are a component of your infrastructure.
- It might be some low level component such as a physical server, virtual machine, or container. Or it can be a higher level component such as an email provider, DNS record, or database provider.
- The resource block creates a resource of the given TYPE (first parameter) and NAME (second parameter). The combination of the type and name must be unique.

Resource Configurations - Example

- A resource configuration looks like the following:

```
resource "aws_instance" "web" {
    ami                  = "ami-408c7f28"
    instance_type = "t1.micro"
}
```

Terraform Commands (CLI)

- Terraform is controlled via a very easy to use command-line interface (CLI).
- Terraform is only a single command-line application: terraform. This application then takes a subcommand such as "apply" or "plan".
- The terraform CLI is a well-behaved command line application. In erroneous cases, a non-zero exit status will be returned. It also responds to -h and --help as you'd most likely expect.

```
terraform --help
```

Terraform Commands (CLI)

- Get Terraform plugins as per configuration:

```
terraform init
```

- Validate Terraform configurations:

```
terraform validate
```

- Validate configurations in Simulation mode:

```
terraform plan
```

- Apply Terraform configurations:

```
terraform apply
```

Terraform Commands (CLI)

- Destroy Terraform configuration:

```
terraform destroy
```

- Save a plan:

```
terraform plan -out=./plan
```

- Apply a plan:

```
terraform apply ./plan
```

- Show plan or state:

```
terraform show
```

AWS Credentials

- For Terraform Access to AWS Account, following details have to fetched:
  - Region - Go to EC2 Dashboard, and find region in Top Right e.g. ap-south-1
  - Access/Secret Key - Go to
    - Account ID -> Security Credentials -> Access Keys -> Create New Access Key
  - AMI ID

Exercise

- Create an AWS EC2 instance using terraform.

State

- Terraform must store state about your managed infrastructure and configuration.
- This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.
- This state is stored by default in a local file named "terraform.tfstate“.
- 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.

Overrides

- Terraform loads all configuration files within a directory and appends them together. Terraform also has a concept of *overrides*, a way to create files that are loaded last and *merged* into your configuration, rather than appended.
- Overrides have a few use cases:
  - Machines (tools) can create overrides to modify Terraform behavior without having to edit the Terraform configuration tailored to human readability.
  - Temporary modifications can be made to Terraform configurations without having to modify the configuration itself.
- Overrides names must be override or end in _override, excluding the extension. Examples of valid override files are override.tf, override.tf.json, temp_override.tf.

Overrides - Example

- Terraform Configuration example.tf:

```
resource "aws_instance" "web" {
        ami = "ami-408c7f28"
}
```

- Override Configuration override.tf:

```
resource "aws_instance" “web" {
        ami = “ami-123456”
}
```

Variables

- Variables serve as parameters for a Terraform module.

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

variable "zones" {
    default = ["us-east-1a", "us-east-1b"]
}
```

Variables

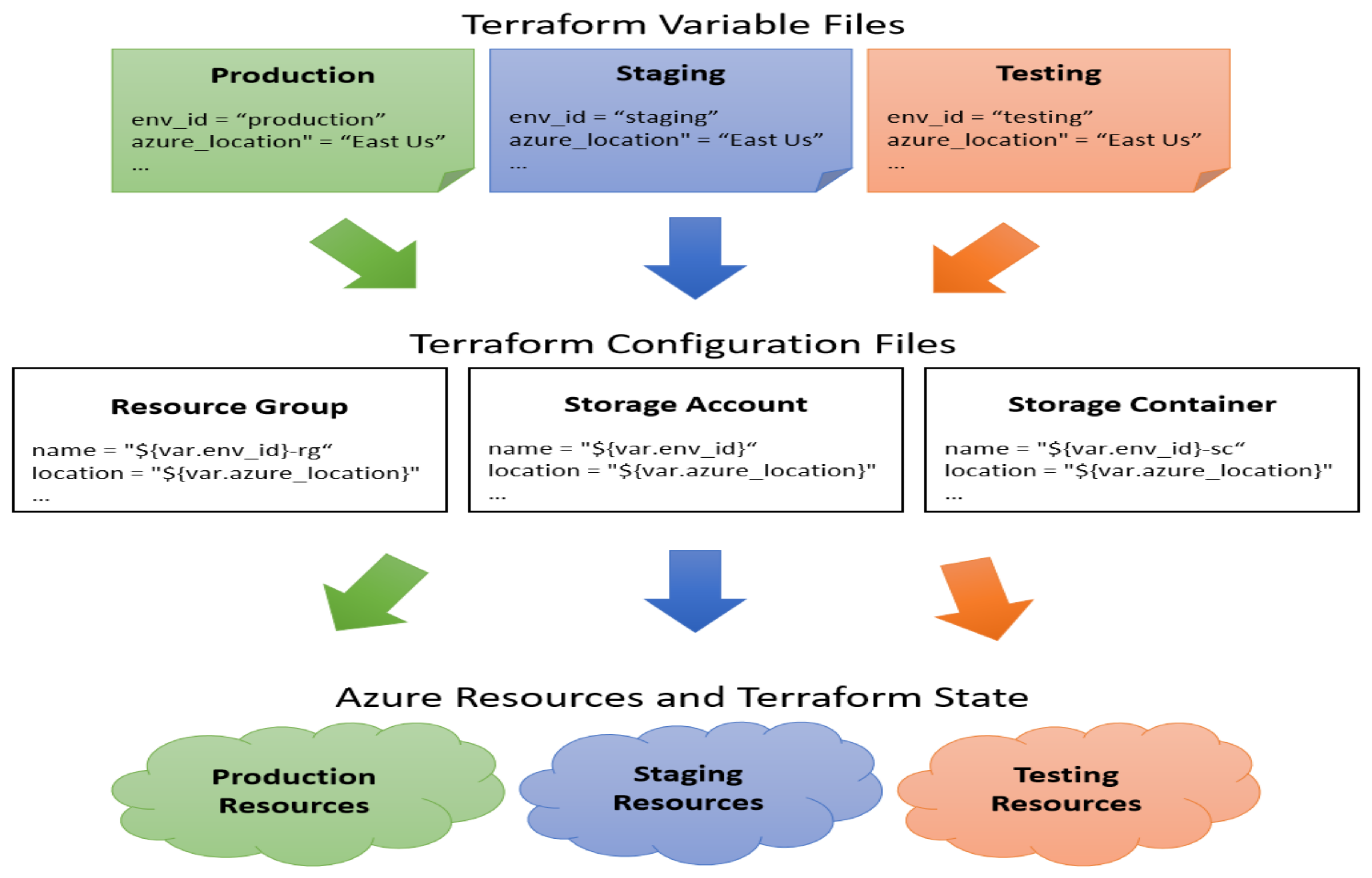
```
variable "list" {
  default = [ "1", "2", "3" ]
}

output "a" {
  value = var.list[0]
}

variable "string" {
  type = string
  default = "Hello, this is sample string"
}

output "b" {
  value = var.string
}
```

Variables

```
variable "number" {
  type = number
  default = 100
}

output "c" {
  value = var.number
}

variable "bool" {
  type = bool
  default = true
}

output "d" {
  value = var.bool
}
```

Variables

```
variable "map" {
  type = map
  default = {name = "Mabel", age = 52}
}

output "e" {
  value = var.map.name
}
```



Terraform Variable Files

Production

```
env_id = “production”
azure_location" = “East Us”
…
```

Staging

```
env_id = “staging”
azure_location" = “East Us”
…
```

Testing

```
env_id = “testing”
azure_location" = “East Us”
…
```

Terraform Configuration Files

Resource Group

```
name = "${var.env_id}-rg“
location = "${var.azure_location}"
…
```

Storage Account

```
name = "${var.env_id}“
location = "${var.azure_location}"
…
```

Storage Container

```
name = "${var.env_id}-sc“
location = "${var.azure_location}"
…
```

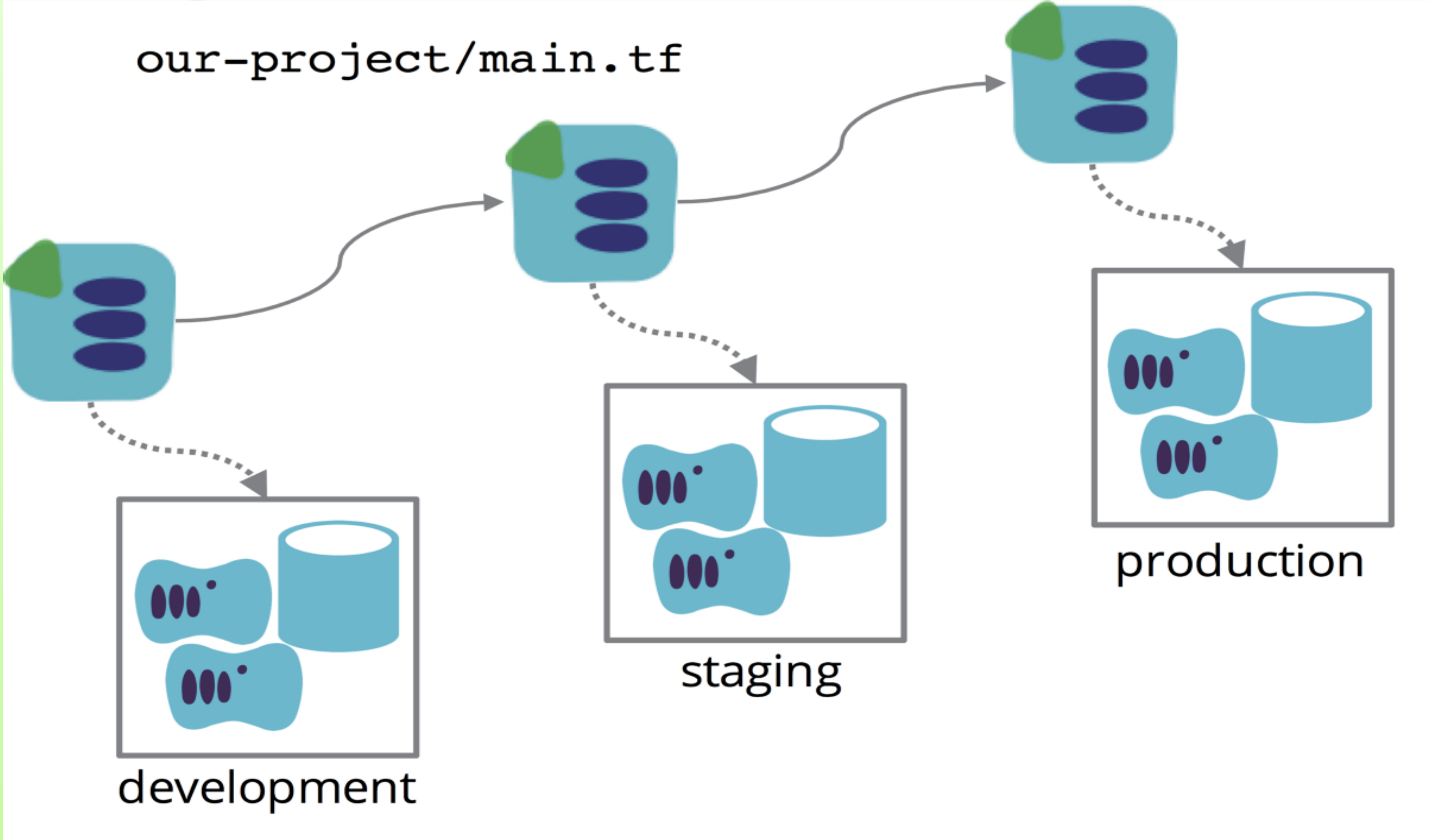
Azure Resources and Terraform State

Production Resources

Staging Resources

Testing Resources

Multiple Environments



Exercise

- Create an AWS EC2 instance using terraform variables defined in variable.tf.

Output Configuration

- Outputs define values that will be highlighted to the user when Terraform applies, and can be queried easily using the output command.
- Terraform knows a lot about the infrastructure it manages. Most resources have attributes associated with them, and outputs are a way to easily extract and query that information.
- For Example,

```
output "address" {
                value = aws_instance.db.public_dns
}
```

- This will output a string value corresponding to the public DNS address of the Terraform-defined AWS instance named "db".

Exercise

- Create an AWS EC2 instance using terraform variables defined in vars.tf and should display Public IP/Public DNS as output item.

Provisioners

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

```
resource "aws_instance" "web" {
            # ...
            provisioner "local-exec" {
                 command = "echo ${self.private_ip} > file.txt"
             }
}
```

Provisioners

- Below are type:
  - Chef
  - Connection
  - File
  - Habitat
  - Local-exec
  - Remote-exec
  - Salt-masterless

Provisioners

```
resource “aws_instance” sample {
  ami = ###


  provisioner "file" {
    source = "test.sh"
    destination = "/tmp/test.sh"
    connection {
      type = "ssh"
      user = "ubuntu"
      private_key = "${file("~/.ssh/id_rsa")}"
      host = self.public_ip
    }
  }

}
```

Provisioners

```
provisioner "remote-exec" {
  inline = [
    "chmod +x /tmp/test.sh",
    "/tmp/test.sh",
  ]
  connection {
    type = "ssh"
    user = "ubuntu"
    private_key = "${file("~/.ssh/id_rsa")}"
    host = self.public_ip
  }
}
```

Exercise

- Create an AWS EC2 instance using terraform variables defined in variable.tf and do ssh login to instance.

Exercise

- Create an AWS EC2 instance using terraform variables defined in variable.tf and should display Welcome Message as "**Welcome to Terraform World!**" on ssh login to instance.

Session: 5

Conditional Statements

Loops

- To accomplish for-loop in terraform, “count” was introduced.
- For Example,

```
resource "aws_instance" "sample" {
          count = 3
          ami = "ami-a523b4dd"
          instance_type = "t2.micro"
          tags = {
              Name = "sample-${count.index}"
           }
}
```

Exercise

- Create a Three AWS EC2 instance using terraform variables defined in variable.tf and using loops.

For Each

- For Example,

```
resource "aws_instance" "sample" {
        for_each = toset( ["blr", "mum", “hyd"] )
        ami = "ami-a523b4dd"
        instance_type = "t2.micro"
        tags = {
            Name = “sample-${each.key}"
         }
}
```

Exercise

- Create a Two AWS EC2 instance with t2.nano and t2.micro using for_each.

If-else

- If-else can be accomplished in terraform, using “count”.
- For Example,

```
variable "create_instance" {
        description = "Create an instance if set to True"
        default = true
}
resource "aws_instance" "example" {
        count = "${var.create_instance == true ? 1 : 0}"
         ami = "ami-a523b4dd"
         instance_type = "t2.micro"
}
```

Exercise

- Create an AWS EC2 instance using terraform variables value.

Session: 6

Terraform Templates

Templates

- The template provider exposes data sources to use templates to generate strings.
- Templates are helpful in scenario of managing multiple environments.

Templates

- For Example,

```
  data "template_file" "user_data" {
     template = "${file("${path.module}/user_data")}"
  }


  resource "aws_instance" "example" {
      instance_type = "${trimspace(data.template_file.user_data.rendered)}"
      ami                = "${var.ami}"
      key_name           = "deployer-key"
}
```

Exercise

- Create Two AWS EC2 instance using terraform variables defined in variable.tf and having different instance_types based on environment prod and dev.

Modules

- A Terraform module is very simple: any set of Terraform configuration files in a folder is a module.
- Here are some of the ways that modules help solve the problems:
  - Organise configuration
  - Encapsulate configuration
  - Re-use configuration
  - Provide consistency

Modules

- Create directory say “terraform_modules”.
- Create Sub-directory inside it “modules/services/webserver-cluster”.
- Create below structure:

```
# tree modules/services/webserver-cluster/
modules/services/webserver-cluster/
├── main.tf
├── outputs.tf
└── vars.tf

0 directories, 3 files
```

Modules - Calling

- Calling Terraform modules:

```
[terraform_modules]# cat main.tf
module "webserver_cluster" {
  source = "./modules/services/webserver-cluster"
  instance_type   = "t2.medium"
}
```

Exercise

- Create Two AWS EC2 instance using terraform variables defined in variable.tf and having different instance_types based on environment prod(t2.micro) and dev(t2.nano) using modules.

Remote State/Backend

- A backend defines where Terraform stores its state data files.
- Below are different types of Backends:
  - local
  - artifactory
  - Azurerm
  - S3
  - Kubernetes
  - http
  - Swift

Remote State/Backend

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


data "terraform_remote_state" "foo" {
   backend = "local"

   config = {
      path = "${path.module}/../../terraform.tfstate"
   }
}
```

Terraform Import

- Terraform 'import' feature lets you down "tfstate" files on existing infrastructure.
- Below are steps:
  - Create "main.tf" with provider and resource to be imported
  - Initialise the plugins
  - Execute **terraform import TYPE.NAME ID**

Terraform Get Data - http

- Terraform can download data from web API as shown below:

```
data "http" "example" {
    url = “https://www.atgensoft.com"
    request_headers = {
       Accept = “application/json”
     }
 }
```

Terraform Random

- The "random" provider allows the use of randomness within Terraform configurations.
- Below is example:

```
resource "random_integer" "num" {
   min = 1
   max = 50000
}
```

- Task: Create a random string of length 16.

Terraform Local

- The "local" provider allows to manage local files.
- Below is example:

```
resource “local_file” “foo” {
  content = “foo”
  filename = “${path.module}/foo.bar”
 }
```

Resource time_sleep & dependency

- The "time" resource allows sleep and depends_on for dependency between resources.
- Below is example:

```
resource "null_resource" "previous" {}
resource “time_sleep” “wait_30_seconds” {
   depends_on = [null_resource.previous]
   create_duration = “30s”
}
resource “null_resource” “next” {
   depends_on = [time_sleep.wait_30_seconds]
}
```

Terraform Taint

- The `terraform taint` command informs Terraform that a particular object has become degraded or damaged. Terraform represents this by marking the object as "tainted" in the Terraform state, and Terraform will propose to replace it in the next plan you create.

```
terraform taint <address>
```

- For example,

```
terraform taint aws_instance.foo
```

Terraform Workspaces

- Terraform starts with a single workspace named "default". This workspace is special both because it is the default and also because it cannot ever be deleted.

```
terraform workspace new dev
terraform workspace select dev
terraform workspace show
```

Exercise

- Create an AWS EC2(Ubuntu 20.04) instance using terraform variables defined in variable.tf, having output Url/IP and Port and running Web Server.
- Do set inbound/outbound rules for same.
  - Apache Web Server: **apache2**
  - Default Port: **80**
  - SSH Port: **22**

- Task: Do this without using provisioners.

Contact us

- Contact @ http://www.atgensoft.com/
- Linkedin: @atgenautomation
- Twitter: @atgenautomation
- FaceBook: @atgenautomation
- YouTube: @atgenautomation
- Email: SAGAR.MEHTA@ATGENSOFT.COM

Thank You !!

Atgen