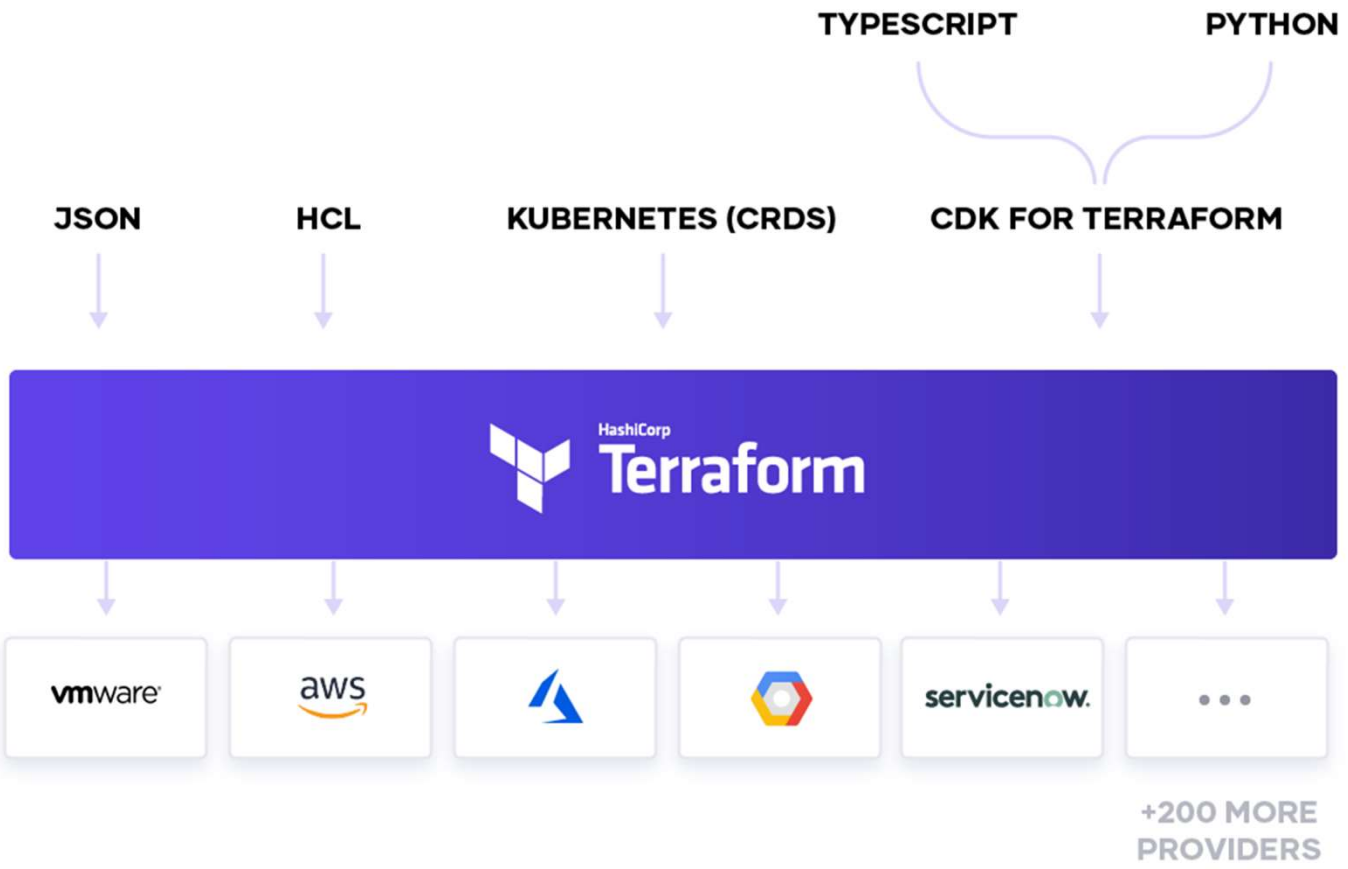
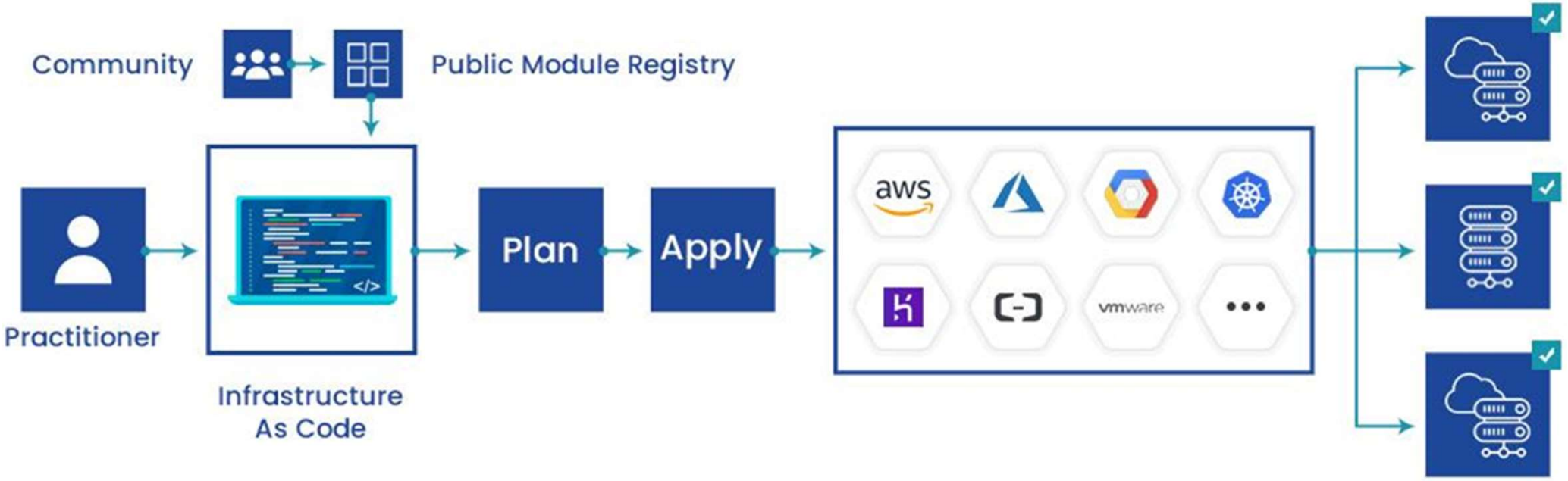
HashiCorp

Terraform

Introduction



Introduction



What is Terraform?

- Tool for
  - Building,
  - Changing, and
  - Versioning infrastructure safely and efficiently

- Can manage
  - Existing and popular service providers as well as
  - Custom in-house solutions.

The key features of Terraform

- Infrastructure as Code
  - Allows a blueprint of datacenter
  - Infrastructure can be shared and re-used.
- Execution Plans
  - Shows what Terraform will do when you call apply
  - Avoid any surprises when Terraform manipulates infrastructure
- Resource Graph
  - Builds a graph of all your resources
  - Parallelizes the creation and modification of any non-dependent resources
  - Builds infrastructure as efficiently as possible
- Change Automation

Why Terraform?

- IAAC
- Platform agnostic
- Enables you to implement all kinds of coding principles
  - Having code in source control
  - Ability to write automated tests
- Community and is open source
- Speed

What is infrastructure as code?

- The process of managing and provisioning computer data centers through machine-readable definition files
- The definitions may be in a version control system
- It can use either scripts or declarative definitions
- More often used to promote declarative approaches.

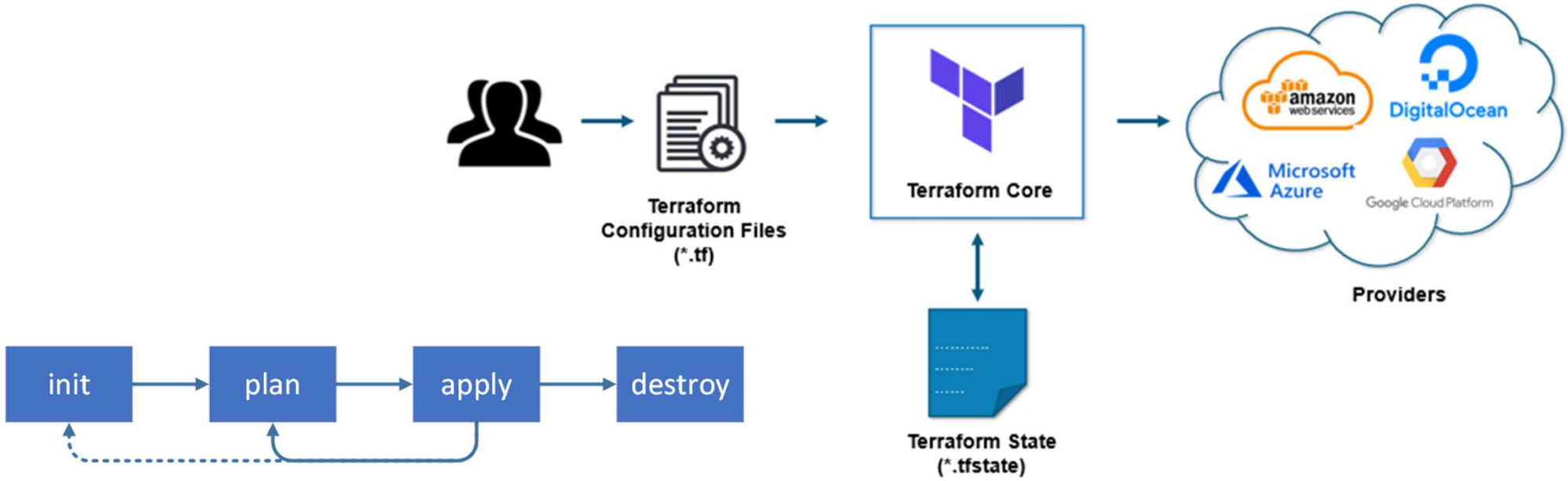
Benefits of infrastructure as code

- Speed and simplicity

- Configuration consistency
  - Standard operating procedures
  - Avoids human error
- Minimization of risk
  - Imagine having a lead engineer be the only one who knows the ins and outs of your infrastructure setup
  - Now imagine that engineer leaving your company.
- Increased efficiency in software development
- Cost savings
  - Spend less time performing manual work
  - Time is money

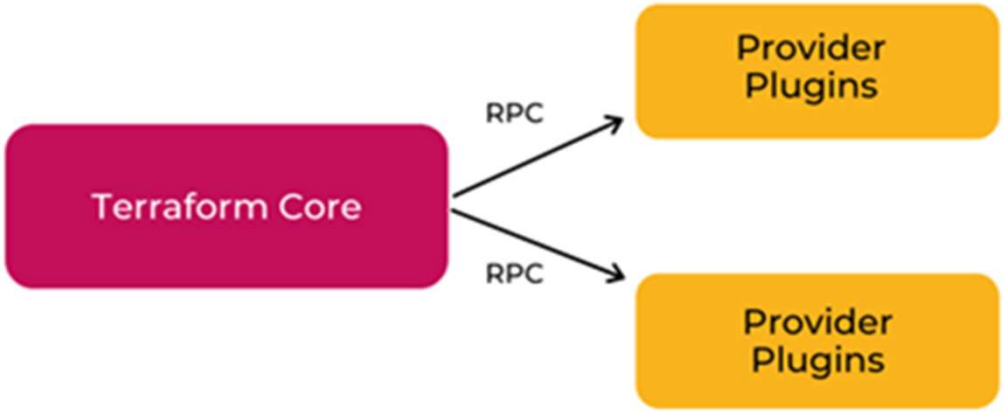
How Terraform works?

- Split into two main parts
  - Terraform Core and
    - Uses remote procedure calls (RPC) to communicate with Terraform Plugins
  - Terraform Plugins
    - Expose an implementation for a specific service, such as AWS



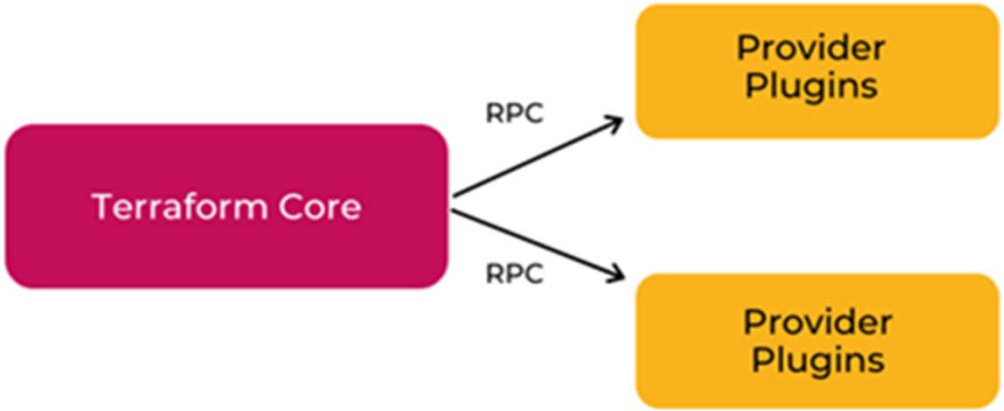
Terraform Core

- Written in the Go programming language
- Compiled binary
  - Is the command line tool (CLI) terraform
- The entrypoint for anyone using Terraform
- Primary responsibilities
  - Reading configuration files
  - Resource state management
  - Construction of the Resource Graph
  - Plan execution
  - Communication with plugins



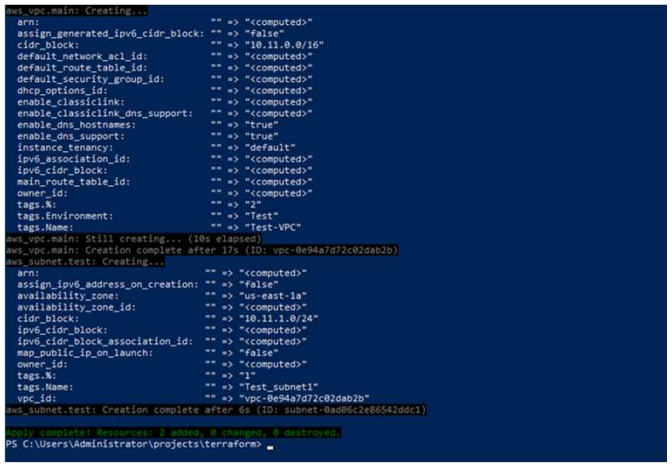
Terraform Plugins

- Written in Go
- Each plugin exposes an implementation for a specific service
  - such as AWS, Azure
- Primary responsibilities
  - Authentication with the Infrastructure Provider
  - Define Resources that map to specific Services
- Plugin Locations
  - ~/.terraform.d/plugins



Install Terraform

- curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add –
- sudo apt-add-repository "deb [arch=amd64] https://apt.releases.hashicorp.com $(lsb_release -cs) main“
- sudo apt-get update && sudo apt-get install terraform
- terraform –help
- terraform -install-autocomplete



Compare to other IAAC tools

	Chef	Puppet	Ansible	SaltStack	Terraform
Cloud		All	All	All	All
Type	Config Mgmt	Config Mgmt	Config Mgmt	Config Mgmt	Orchestration
Infrastructure	Mutable	Mutable	Mutable	Mutable	Immutable
Language	Procedural	Declarative	Procedural	Declarative	Declarative
Architecture	Client/Server	Client/Server	Client only	Client only	Client only
Orchestration					
Lifecycle (state) management	No	No	No	No	Yes
VM provisioning	Partial	Partial	Partial	Partial	Yes
Networking	Partial	Partial	Partial	Partial	Yes
Storage Management	Partial	Partial	Partial	Partial	Yes

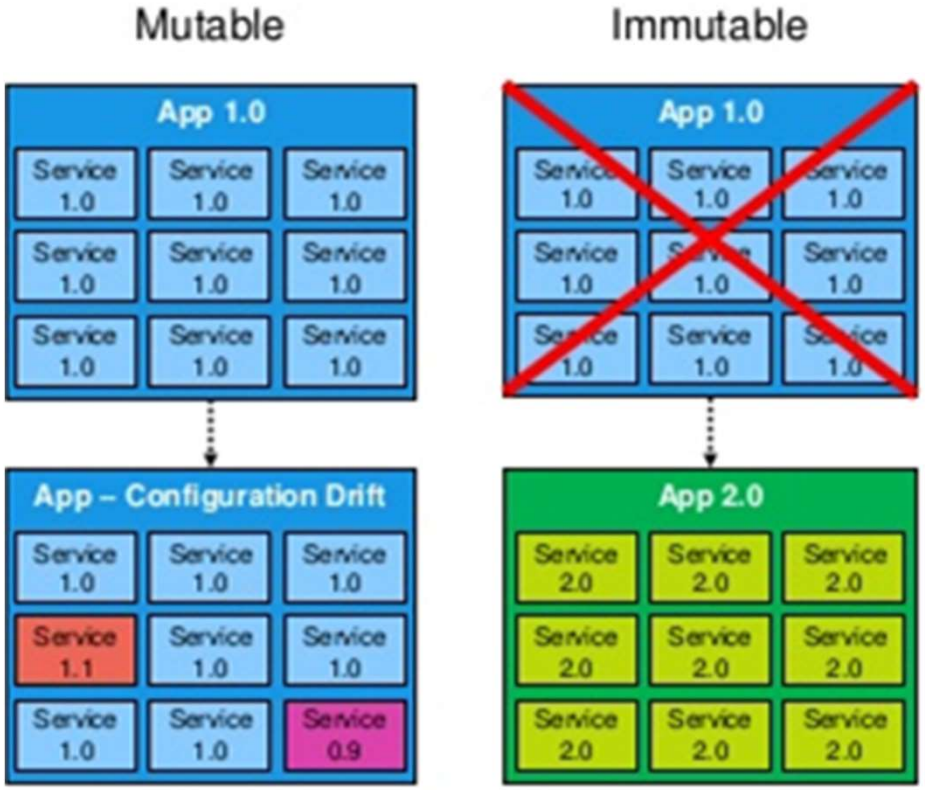
Configuration Management vs Provisioning

- Configuration management tools
  - Designed to install and manage software on existing servers
- Provisioning tools
  - Designed to provision the servers themselves
  - As well as the rest of your infrastructure, like load balancers, databases, networking configuration, etc
  - Leave the job of configuring those servers to other tools

Mutable Infrastructure vs Immutable Infrastructure

- Example: To deploy new version of OpenSSL, would create a new image with the new version of OpenSSL,
- Deploy that image across a set of totally new servers, and
- Then undeploy the old servers

- Makes it easier to know exactly what software is running on a server
- Allows to deploy any previous version of the software at any time



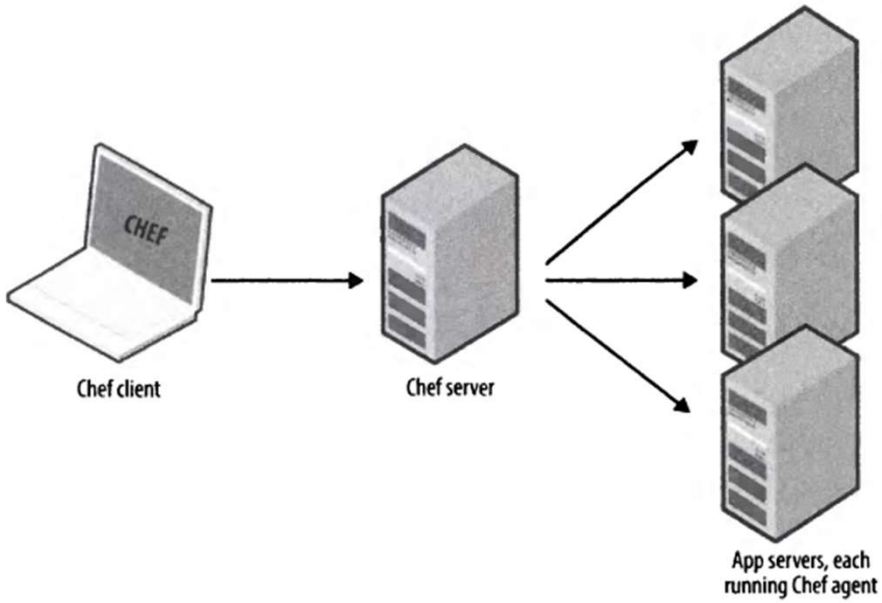
Configuration management vs orchestration

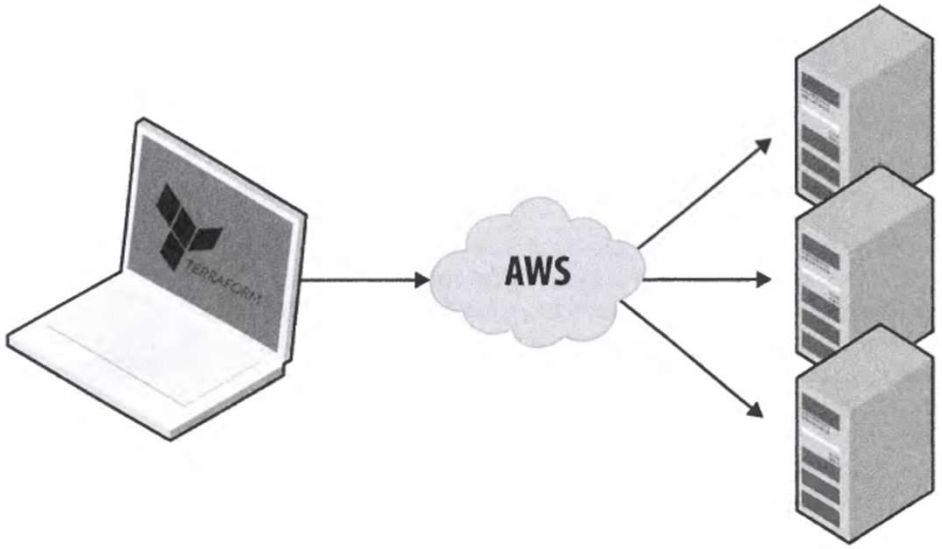
- Orchestration
  - Terraform and AWS CloudFormation
  - Arranging or coordinating multiple systems
  - It's also used to mean "running the same tasks on a bunch of servers at once, but not necessarily all of them."
  - Designed to automate the deployment of servers
- Configuration Management
  - Chef, Puppet
  - Help configure the software and systems on this infrastructure that has already been provisioned

Procedural language vs declarative language

- With Terraform's declarative approach, the code always represents the latest state of your infrastructure.
- Focus on describing your desired state

Agent vs Agentless





An Introduction to Terraform Syntax

- Called HashiCorp Configuration Language (HCL)
- Human readable as well as machine-friendly

```
# An AMI
variable "ami" {
  description = "the AMI to use"
}

resource "aws_instance" "web" {
  ami           = "${var.ami}"
  count         = 2
  source_dest_check = false
  connection {
    user = "root"
  }
}
```

How to create reusable infrastructure

- Module basics
- Module inputs
- Module outputs
- Versioned modules

Module basics

- Any set of Terraform configuration files in a folder is a module.
  - $ tree minimal-module/
  - .
  - ├── README.md
  - ├── main.tf
  - ├── variables.tf
  - ├── outputs.tf

- At least one module, known as its root module
  - Consists of the resources defined in the .tf files in the main working directory.
- A module can call other modules

Calling a Child Module

- module "servers" {
- source = "./app-cluster"

- servers = 5
- }

Module inputs

- Parameters for a Terraform module
- Like function arguments
  - variable "image_id" {
  - type = string
  - }

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

Using Input Variable Values

- var.<NAME>

- resource "aws_instance" "example" {
- instance_type = "t2.micro"
- ami = var.image_id
- }

Variable Definitions (.tfvars) Files

- To set lots of variables, it is more convenient to specify their values in a variable definitions file
- And then specify that file on the command line with -var-file
  - terraform apply -var-file="testing.tfvars"

Module outputs

- Like the return values of a Terraform module
  - output "instance_ip_addr" {
  - value = aws_instance.server.private_ip
  - }

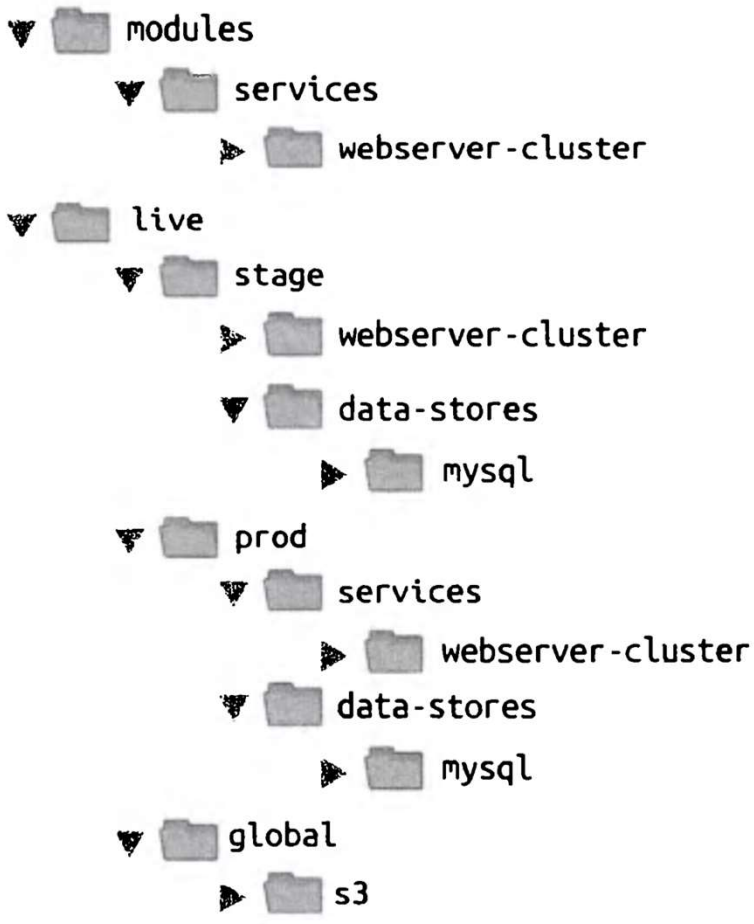
Accessing Child Module Outputs

- In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>

- For example, if a child module named web_server declared an output named instance_ip_addr, you could access that value as
  module.web_server.instance_ip_addr

Versioned modules

- Both staging and production are pointing to the same module folder?
- Code changes will affect both environments
- Harder to test a change in staging without affecting production
- Create versioned modules so that you can use one version in staging



Terraform loops, if-statements and deployment

- Loops
- If-statements
- If-else-statements
- Zero-downtime deployment

Loops

- resource "aws_lam_user" "example" {
- count = 3
- name = "neo.${count.Index}"
- }

If-statements

- Terraform doesnt support if-statements, so this code won't work
- However, you can accomplish the same thing by using the count parameter
  - resource "aws_autoscallng_schedule" "scale_out_durlng_buslness_hours" {
  - count = "${var.enable_autoscaling}"
  - scheduled_action_name = "scale-out-durlng-buslness-hours"
  - niin_size = 2
  - nax_size = 10
  - desired_capacity = 10
  - recurrence ="09** *"
  - autoscaling_group_name = "${aws_autoscaling_group.example.name}"
  - }

If-else-statements

- resource "aws_lam_user_pollcy_attachment" "neo_cloudwatch_full_accessM {
- count = "${var.glve_neo_cloudwatch_full_access}"
- user = "${aws_lam_user.example.0.name}"
- pollcy_arn = "${aws_lam_pollcy.cloudwatch_full_access.arn}"
- }
- resource Maws_lam_user_pollcy_attachmentM Mneo_cloudwatch_read_only" {
- count = "${1 - var.glve_neo_cloudwatch_full_access}"
- user = "${aws_iam_user. example. 0. name}11
- pollcy_arn = M${aws_lam_pollcy.cloudwatch_read_only.arn}"
- }

Zero-downtime deployment

- Problem 1: How to ensure new infrastructure is created before the old is destroyed

- Problem 2: A running VM does not necessarily mean a working application

Ensure new infrastructure is created before

```
resource "digitalocean_droplet" "web" {
  count  = 2
  image  = "${var.image}"
#...

  lifecycle {
    create_before_destroy = true
  }
}
```

A running VM not necessarily mean working

```
resource "digitalocean_droplet" "web" {
  count  = 2
  image  = "${var.image}"
#...
  lifecycle {
    create_before_destroy = true
  }

  provisioner "local-exec" {
    command = "./check_health.sh ${self.ipv4_address}"
  }
}
```

Getting Started & Setting Up Labs

- Install Terraform
- Choosing Right IDE for Terraform
  - - VSCode
- Use AWS account

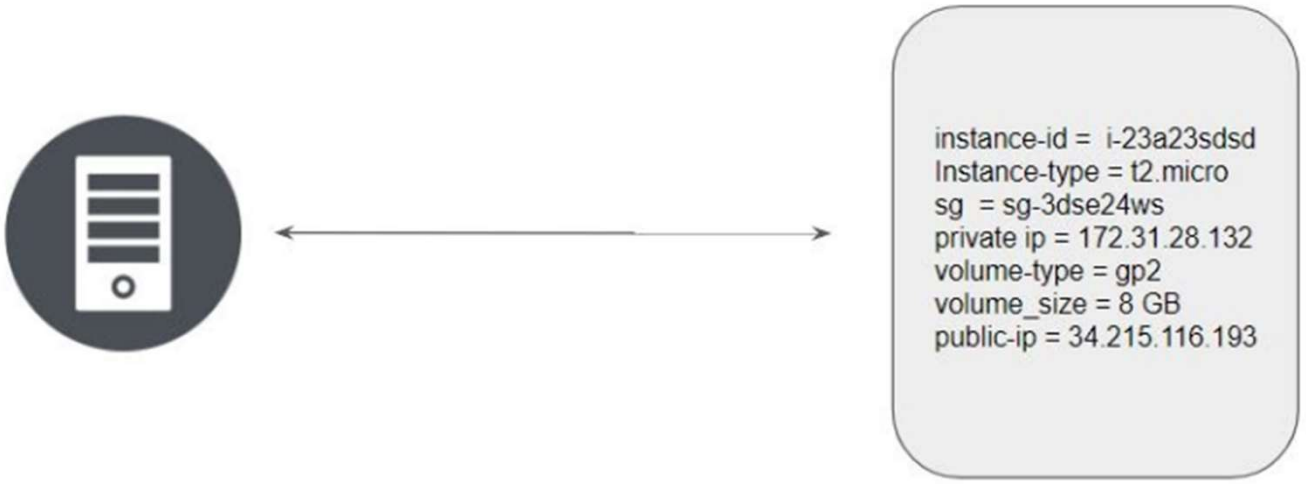
Deploying Infrastructure with Terraform

Hands-On

- Refer:
  - 3 - Deploying Infrastructure\1-first_ec2

TFState file

- Terraform stores the state of the infrastructure that is being created from the TF files.
- This state allows terraform to map real-world resource to your existing configuration.



Current State vs Desired State

- Current State
  - Current Infrastructure Resource & Configuration
- Desired State
  - Infrastructure Configuration defined within the Terraform TF Files

- Terraform will plan to match the desired state to the current state
- If there is a difference between both, the desired state will take the preference.

Read, Generate, Modify Configurations

Attributes and Output Values

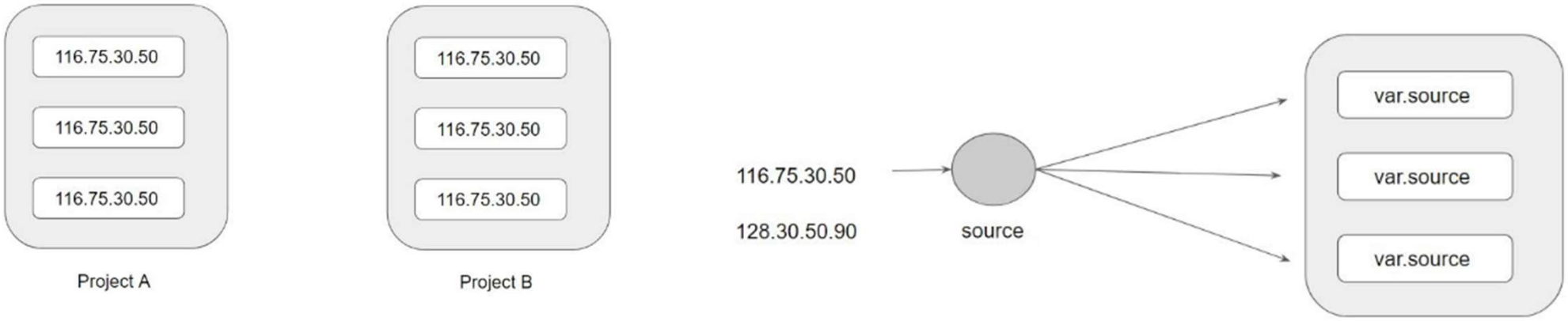
- Terraform has the capability to output the attribute of a resource with the output values.
  - Example:
    - ec2_public_ip = 35.161.21.197
    - bucket_identifier = terraform-test-kplabs.s3.amazonaws.com
- An outputed attributes can not only be used for the user reference but it can also act as an input to other resources being created via terraform
- After EIP gets created, it's IP address should automatically get whitelisted in the security group.

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\1-attributes
  - 02 - Read, Generate, Modify Configurations\2-reference

Terraform variables

- Repeated static values can create more work in the future.



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

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\3-variables

Approach for Variable Assignment

- Variables in Terraform can be assigned values in multiple ways.
- Some of these include:
  - Environment variables
    - export TF_VAR_instancetype="t2.nano"
    - echo $TF_VAR
  - Command Line Flags
    - terraform plan -var="instancetype=t2.small"
    - terraform plan -var-file="custom.tfvars"
  - From a File
    - instancetype="t2.large"
  - Variable Defaults
    - variable "instancetype" {
    - default = "t2.micro"
    - }

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\4-approach-to-variable-assignment

Data Types for Variables

- variable "image_id" {
- type = string
- }
- If no type constraint is set then a value of any type is accepted.

Type Keywords	Description
string	Sequence of Unicode characters representing some text, like "hello".
list	Sequential list of values identified by their position. Starts with 0 [“mumbai” ,”singapore”, ”usa”]
map	a group of values identified by named labels, like {name = "Mabel", age = 52}.
number	Example: 200

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\5-data-types

Count Parameter

- Can simplify configurations and let you scale resources by simply incrementing a number.
- Let’s assume
  - You need to create two EC2 instances
  - One of the common approaches is to define two separate resource blocks for aws_instance.

```
resource "aws_instance" "instance-1" {
    ami = "ami-082b5a644766e0e6f"
    instance_type = "t2.micro"
    count = 5
}
```

Count Index

- With the below code, terraform will create 5 IAM users. But the problem is that all will have the same name.

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  count = 5
  path = "/system/"
}
```

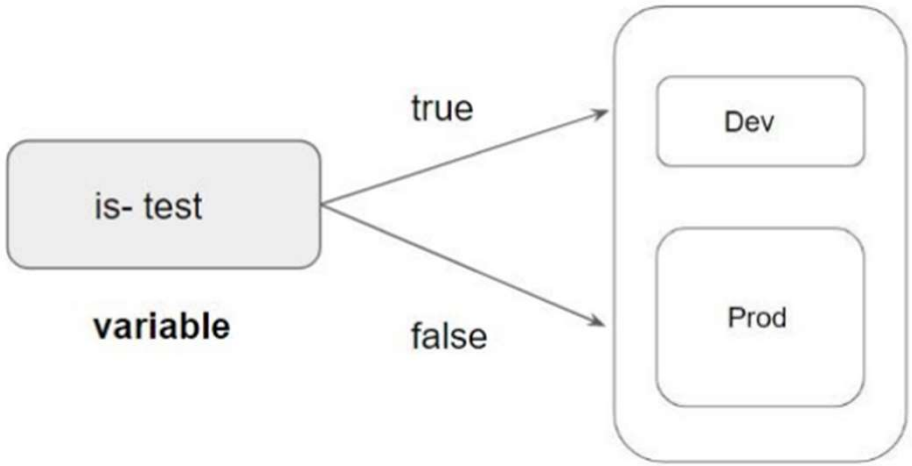
- count.index allows us to fetch the index of each iteration in the loop.

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\6-Count and Count Index

Conditional Expression

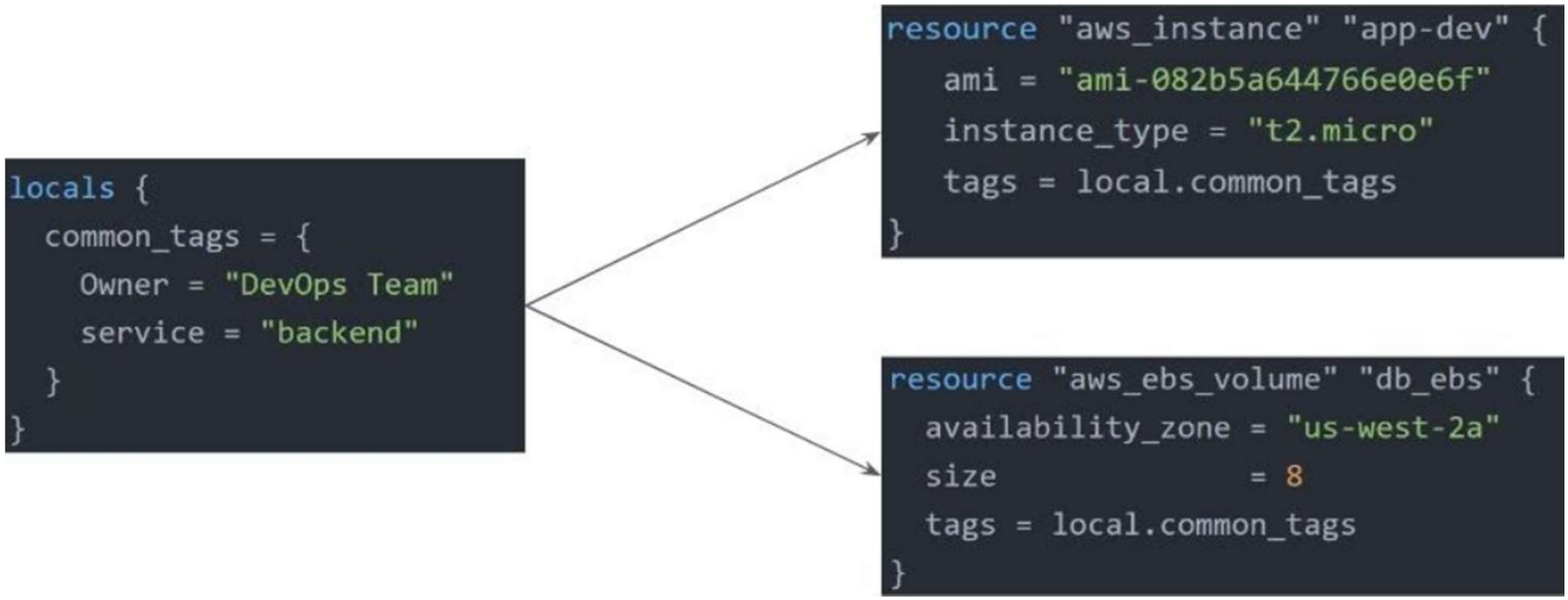


Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\7-Conditional Expressions

Local Values

- A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.



Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\8-Local Values

Terraform Functions

- Built-in functions to use
- Example:
  - max(5, 12, 9)
- The Terraform language does not support user-defined functions

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\9-Terraform Functions

Dynamic Blocks

- In many of the use-cases, there are repeatable nested blocks that need to be defined.
- This can lead to a long code and it can be difficult to manage in a long time.

```
ingress {
  from_port   = 9200
  to_port     = 9200
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

```
ingress {
  from_port   = 8300
  to_port     = 8300
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

Dynamic Blocks

- Dynamic Block allows us to dynamically construct repeatable nested blocks

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Overview of Iterators

- The iterator argument (optional) sets the name of a temporary variable that represents the current element of the complex value
- If omitted, the name of the variable defaults to the label of the dynamic block ("ingress" in the example above).

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```
dynamic "ingress" {
  for_each = var.ingress_ports
  iterator = port
  content {
    from_port   = port.value
    to_port     = port.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Hands-On

- Refer
  - 02 - Read, Generate, Modify Configurations\10-dynamic-block

Modules

- A container for multiple resources that are used together
- Modules can be used to create lightweight abstractions

- $ tree minimal-module/
- .
- ├── README.md
- ├── main.tf
- ├── variables.tf
- ├── outputs.tf

Modules

- $ tree complete-module/
  - ├── README.md
  - ├── main.tf
  - ├── variables.tf
  - ├── outputs.tf
  - ├── ...
  - ├── modules/
  - │   ├── nestedA/
  - │   │   ├── README.md
  - │   │   ├── variables.tf
  - │   │   ├── main.tf
  - │   │   ├── outputs.tf
  - │   ├── nestedB/
  - │   ├── .../
  - ├── examples/
  - │   ├── exampleA/
  - │   │   ├── main.tf
  - │   ├── exampleB/
  - │   ├── .../

Calling a Child Module

- module "servers" {
- source = "./app-cluster"

- servers = 5
- }

- Refer:
  - Terraform\modules

Terraform Provisioners

Understanding the Challenge

- Till now we have been working only on the creation and destruction of infrastructure scenarios.
- Let’s take an example:
  - We created a web-server EC2 instance with Terraform.
  - Problem: It is only an EC2 instance, it does not have any software installed.

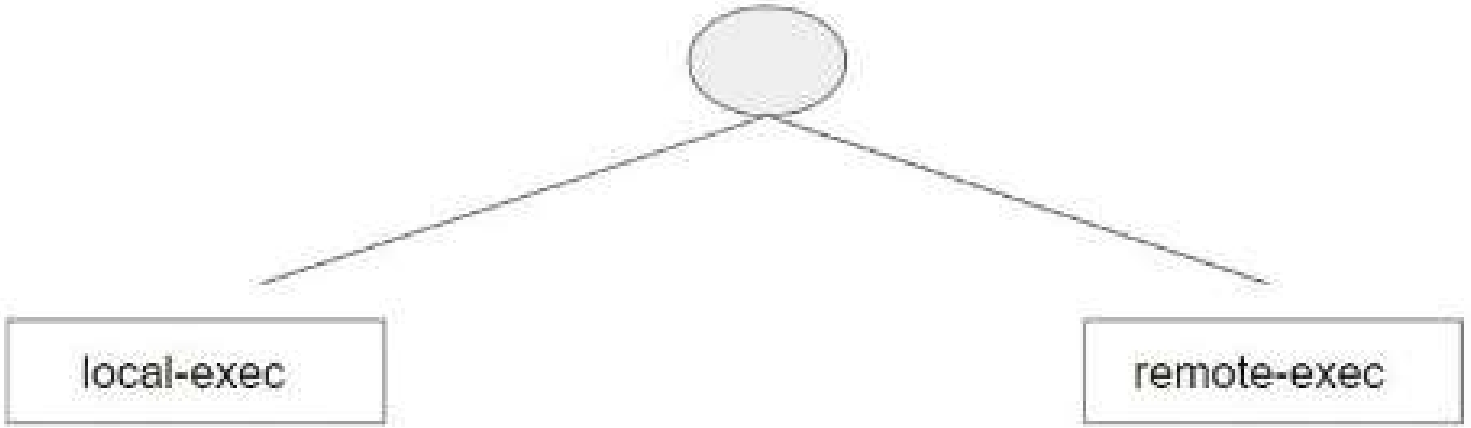
- What if we want a complete end to end solution?

Terraform Provisioners

- Used to execute scripts on a local or remote machine as part of resource creation or destruction.
- Let’s take an example:
  - On creation of Web-Server, execute a script which installs Nginx web-server.

Types of Provisioners

- Terraform has the capability to turn provisioners both at the time of resource creation as well as destruction.
- There are two main types of provisioners:



Local Exec Provisioners

- Allow us to invoke a local executable after the resource is created.

- One of the most used approaches of local-exec is to run ansible-playbooks on the created server after the resource is created.

  - provisioner "local-exec" {
  - command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  - }

Remote Exec Provisioners

- Allow invoking scripts directly on the remote server.

```
resource "aws_instance" "web" {
  # …

  provisioner "remote-exec" {
                    ……..
  }
}
```

Provisioner Types

- There are two primary types of provisioners:

Types of Provisioners	Description
Creation-Time Provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle If a creation-time provisioner fails, the resource is marked as tainted.
Destroy-Time Provisioner	Destroy provisioners are run before the resource is destroyed.

Provisioner Types

- If when = destroy is specified, the provisioner will run when the resource it is defined within is destroyed.

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

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

Failure Behavior

- By default, provisioners that fail will also cause the terraform apply itself to fail.
- The on_failure setting can be used to change this. The allowed values are:

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

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

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

Terraform Modules & Workspaces

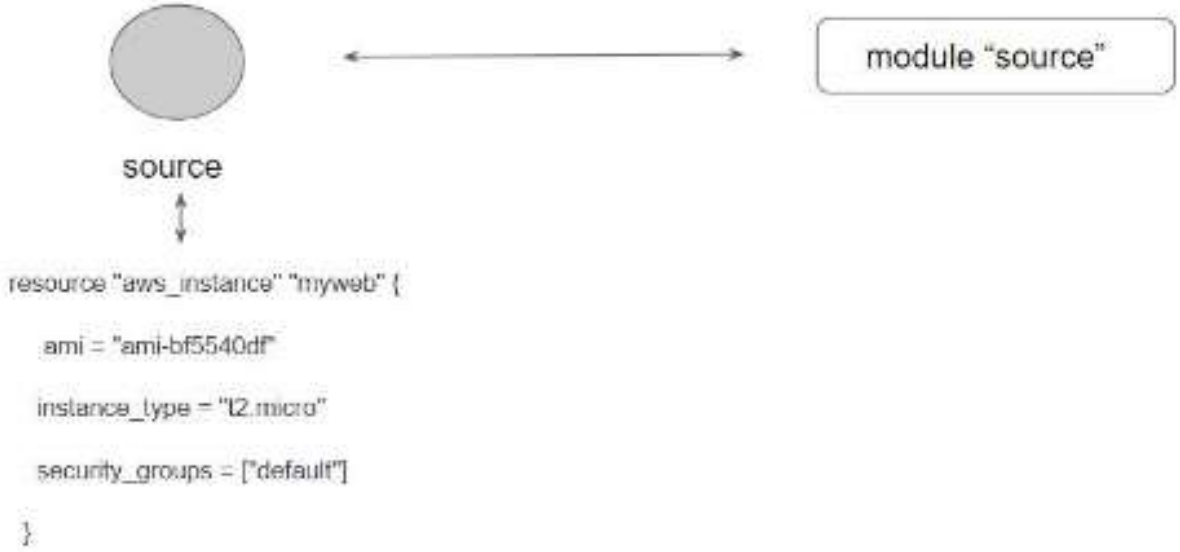
Generic Scenario:

- We do repeat multiple times various terraform resources for multiple projects.
- Sample EC2 Resource
  - resource "aws_instance" "myweb" {
  - ami = "ami-bf5540df"
  - instance_type = "t2.micro"
  - security_groups = ["default"]
  - }

- Instead of repeating a resource block multiple times, we can make use of a centralized structure.

Centralized Structure

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



Challenges with Terraform Modules

- One common need for infrastructure management is to build environments like
  - Staging
  - Production
- So how do we keep environment variables different

Terraform Registry

- A repository of modules written by the Terraform community

Module Location

- If we intend to use a module, we need to define the path where the module files are present.
- The module files can be stored in multiple locations, some of these include:
  - Local Path
  - GitHub
  - Terraform Registry
  - S3 Bucket
  - HTTP URLs

Verified Modules in Terraform Registry

- Are maintained by various third-party vendors.
- Are available for various resources like AWS VPC, RDS, ELB, and others
- Are reviewed by HashiCorp
- Actively maintained by contributors to stay up-to-date
- The blue verification badge appears next to modules that are verified.

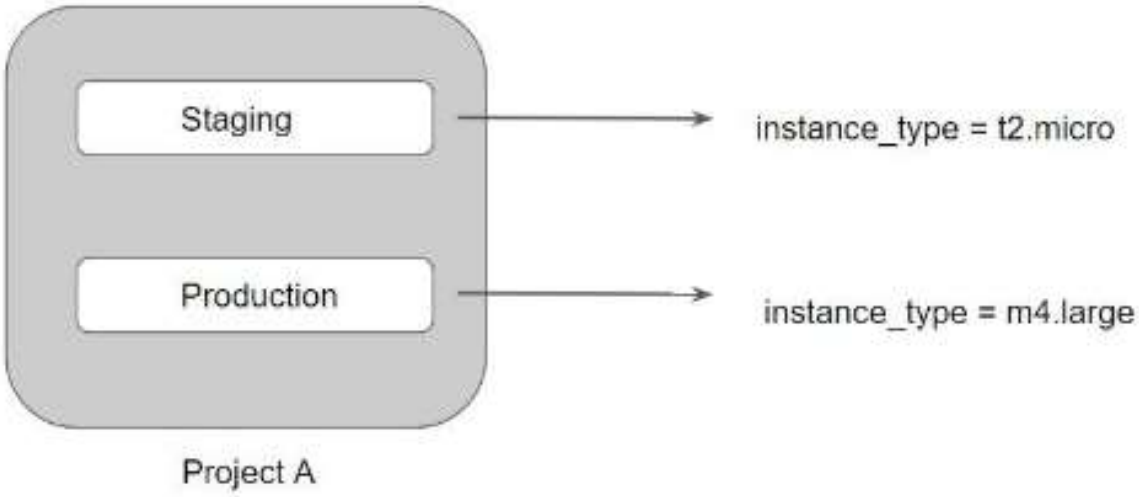
Using Registry Modules in Terraform

- Make use of the source argument that contains the module path.

- Below code references to the EC2 Instance module within terraform registry.
  - module "ec2-instance" {
  - source = "terraform-aws-modules/ec2-instance/aws"
  - version = "2.13.0"
  - # insert the 10 required variables here
  - }

Terraform Workspace

- Terraform allows us to have multiple workspaces, with each of the workspaces we can have a different set of environment variables associated

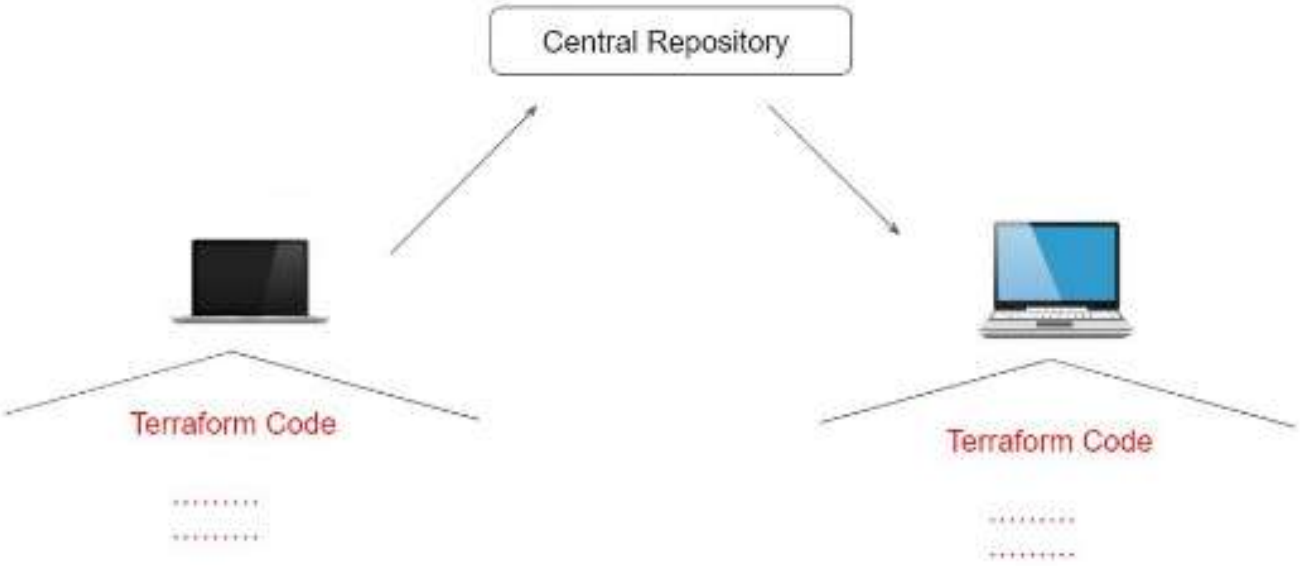


- Terraform starts with a single workspace named "default".

Remote State Management

Integrating with GIT for team management

- Till now, we have been working with terraform code locally.
- Consider the scenario were other members of the team are also working on Terraform.



Module Sources in Terraform

- The source argument in a module block tells Terraform where to find the source code for the desired child module.
  - Local paths
  - Terraform Registry
  - GitHub
  - Bitbucket
  - Generic Git, Mercurial repositories
  - HTTP URLs
  - S3 buckets
  - GCS buckets

Local Path

- A local path must begin with either ./ or ../ to indicate that a local path is intended.

```
module "consul" {
  source = "../consul"
}
```

Git Module Source

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

Terraform & GitIgnore

- The .gitignore file is a text file that tells Git which files or folders to ignore in a project

.gitignore
conf/
*.artifacts
credentials

- Depending on the environment, it is recommended to avoid committing certain files to GIT.

Files to Ignore	Description
.terraform	This file will be recreated when terraform init is run.
terraform.tfvars	Likely to contain sensitive data like usernames/passwords and secrets.
terraform.tfstate	Should be stored in the remote side.
crash.log	If terraform crashes, the logs are stored to a file named crash.log

Remote State Management

- Supports various types of remote backends
  - Standard BackEnd Type: State Storage and Locking
  - Enhanced BackEnd Type: All features of Standard + Remote Management

- In the ideal scenario
  - Terraform configuration code should be part of the centralized Git repositories
  - State file should be part of the remote backends

Remote State Management

- During hands-on, we will use of S3 Backend to store our state file
- Following is the sample configuration file for the S3 backend:

```
terraform {
  backend "s3" {
    bucket = "  -remote-backend"
    key    = "remotedemo.tfstate"
    region = "us-west-1"
    access_key = "YOUR-ACCESS-KEY"
    secret_key = "YOUR-SECRET-KEY"
    dynamodb_table = "s3-state-lock"
  }
}
```

State File Locking

- Whenever you are performing a write operation, terraform would lock the state file.
- This is very important as otherwise during your ongoing terraform apply operations, if others also try for the same, it would corrupt your state file.
- Example:
  - Person A is terminating the RDS resource which has associated rds.tfstate file
  - Person B has now tried resizing the same RDS resource at the same time.

- For the S3 backend, you can make use of the DynamoDB for state file locking functionality.

Terraform State Management

State Sub Command	Description
list	List resources within terraform state file.
mv	Moves item with terraform state.
pull	Manually download and output the state from remote state.
push	Manually upload a local state file to remote state.
rm	Remove items from the Terraform state
show	Show the attributes of a single resource in the state.

Sub Command - List

- The terraform state list command is used to list resources within a Terraform state

```
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.webapp
```

Sub Command - Move

- To move items in a Terraform state.
- To rename an existing resource without destroying and recreating it.
  - terraform state mv [options] SOURCE DESTINATION

Sub Command - Pull

- To manually download and output the state from a remote state.
- This is useful for reading values out of state

Sub Command - Remove

- To remove items from the Terraform state.
- Not physically destroyed.
- Are no longer managed by Terraform

- For example
  - If you remove an AWS instance from the state, the AWS instance will continue running, but terraform plan will no longer see that instance.

Sub Command - Show

- Used to show the attributes of a single resource in the Terraform state.

```
bash-4.2# terraform state show aws_instance.webapp
# aws_instance.webapp:
resource "aws_instance" "webapp" {
    ami                          = "ami-082b5a644766e0e6f"
    arn                          = "arn:aws:ec2:us-west-2:018721151861:instance/i-0107ea9ed06c467e0"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2b"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
    get_password_data            = false
    id                           = "i-0107ea9ed06c467e0"
    instance_state               = "running"
    instance_type                = "t2.micro"
```

Terraform Import

- It might happen that there is a resource that is already created manually.
- In such a case, any change you want to make to that resource must be done manually.

Terraform Import

- Able to import existing infrastructure
- Allows to take resources created by some other means and bring it under Terraform management.
- Terraform can only import resources into the state
- It does not generate configuration
- Prior to running terraform import it is necessary to write manually a resource configuration block for the resource, to which the imported object will be mapped.

Security

Provider Configuration

- Till now, we have been hardcoding the aws-region parameter within the providers.tf
- This means that resources would be created in the region specified in the providers.tf file.

Provider Configuration

- To select an aliased provider for a resource, set its provider meta-argument to a <PROVIDER NAME>.<ALIAS> reference:

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

  # ...
}
```

Handling Multiple AWS Profiles in Terraform

- You can define multiple configurations for the same provider, and select which one to use

```
# The default provider configuration
provider "aws" {
  region = "us-east-1"
}

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

Sensitive Parameter

- When working with a field that contains information likely to be considered sensitive, it is best to set the Sensitive property on its schema to true

```
output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database.
  sensitive   = true
}
```

- Setting the sensitive to “true” will prevent the field's values from showing up in CLI output and in Terraform Cloud

```
C:\Users\        \Desktop\terraform\sensitive data>terraform apply

Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

db_password = <sensitive>
```

- It will not encrypt or obscure the value in the state, however.

Terraform Cloud & Enterprise Capabilities

Overview of Terraform Cloud

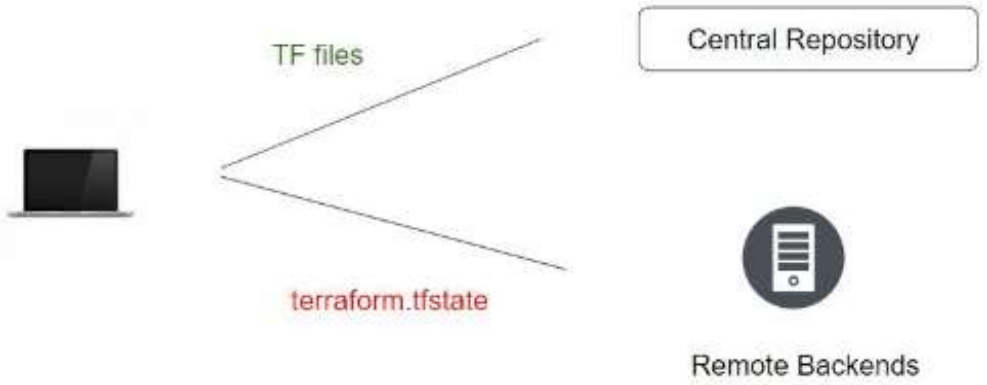
- Manages Terraform runs in a consistent and reliable environment
- Has various features like
  - access controls
  - private registry for sharing modules
  - policy controls etc
- Available as a hosted service at https://app.terraform.io

Overview of Sentinel

- Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products.
- It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.
- Note: Sentinel policies are paid feature

Remote Backends

- Supports various types of remote backends which can be used to store state data.
- As of now, we were storing state data in local and GIT repository.
- Depending on remote backends that are being used, there can be various features.
  - Standard BackEnd Type: State Storage and Locking
  - Enhanced BackEnd Type: All features of Standard + Remote Management



Remote Backends

- When using full remote operations, operations like terraform plan or terraform apply can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal.
- Remote plans and applies use variable values from the associated Terraform Cloud workspace.
- Terraform Cloud can also be used with local operations, in which case state is stored in the Terraform Cloud backend


thanks