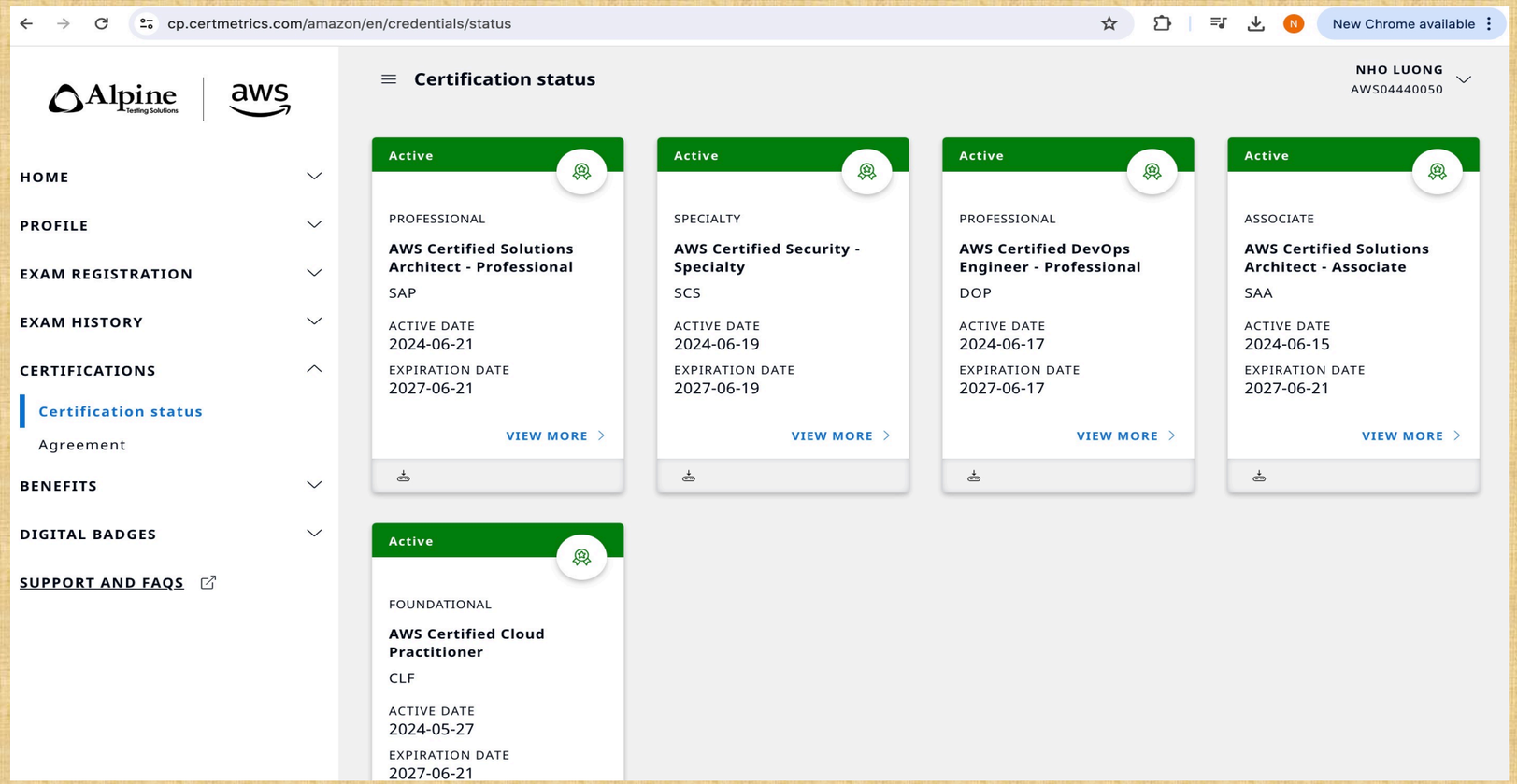
Terraform on AWS Associate

Author: Nho Luong

Skill: DevOps Engineer Lead



TECHSTACK

DevOps & SRE Roadmap

AWS EKS
Kubernetes

Azure AKS
Kubernetes

AWS ECS
Docker on AWS

AWS
CloudFormation

Google GKE
Kubernetes

AWS
Lambda & Serverless

Terraform Associate

Vault & Consul

CKA Certified Kubernetes Administrator

DevOps on AWS
EC2, ECS, EKS, Lambda, VPC

DevOps on Azure
VMs, ACI, AKS, Functions, App Services

SRE with Terraform on AWS
EC2, ECS, EKS, Lambda, VPC

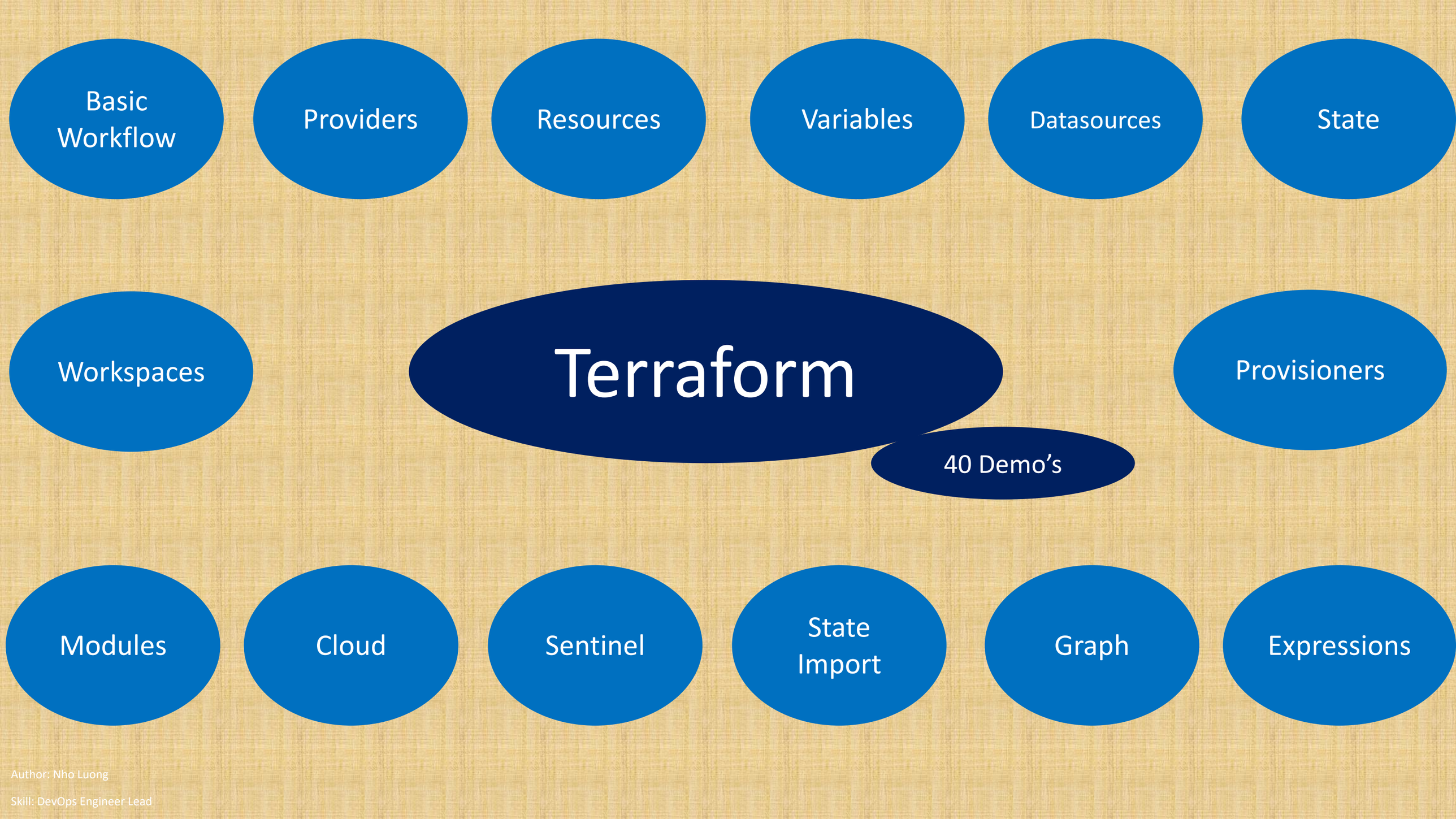
SRE with Terraform on Azure
VMs, ACI, AKS, Functions, App Services

CKAD Certified Kubernetes Application Developer

CKS Certified Kubernetes Security

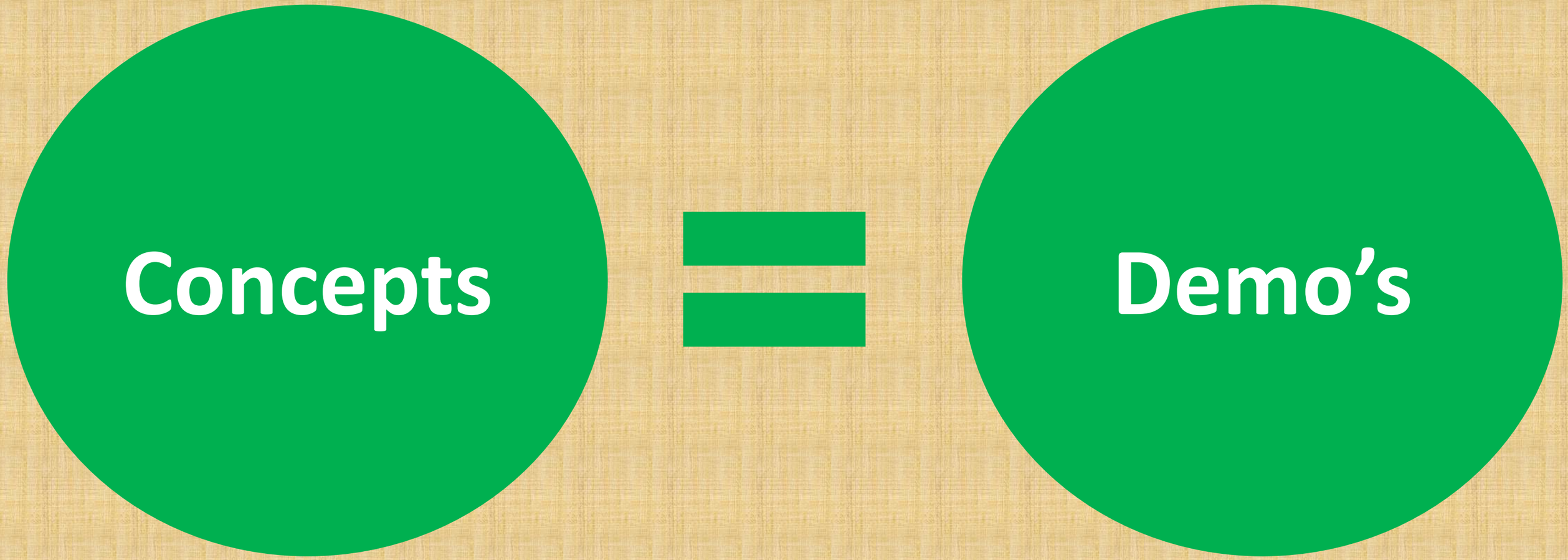
Author: Nho Luong

Skill: DevOps Engineer Lead



Author: Nho Luong

Skill: DevOps Engineer Lead

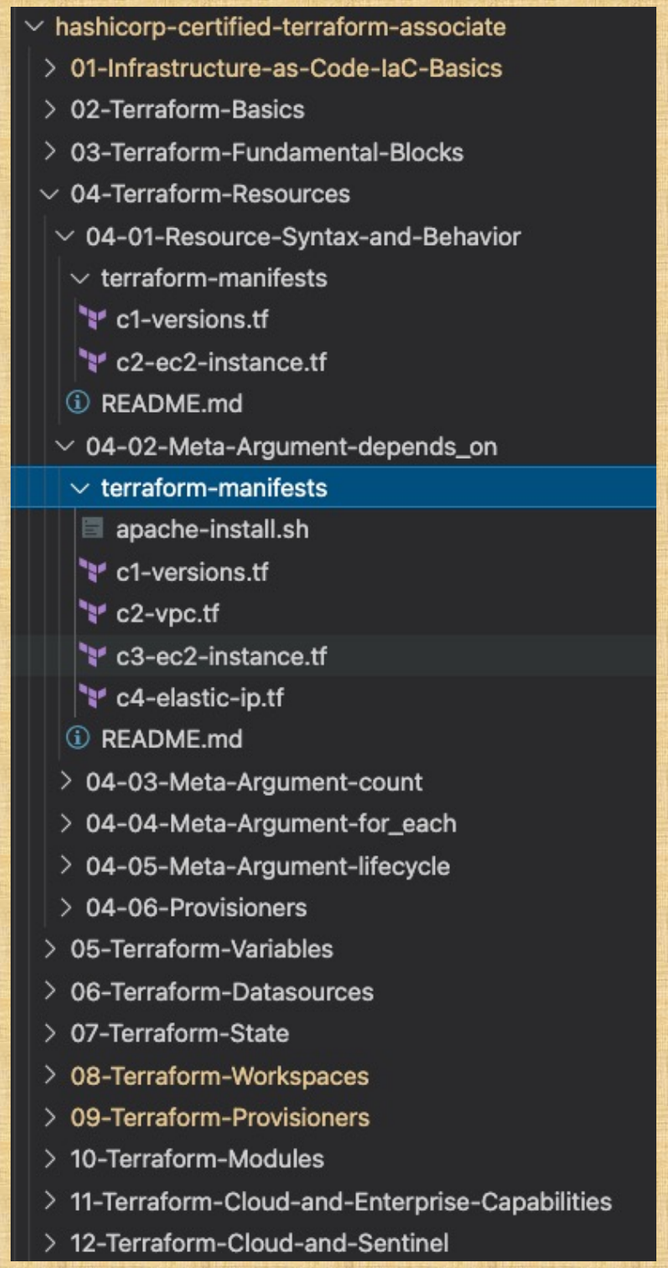


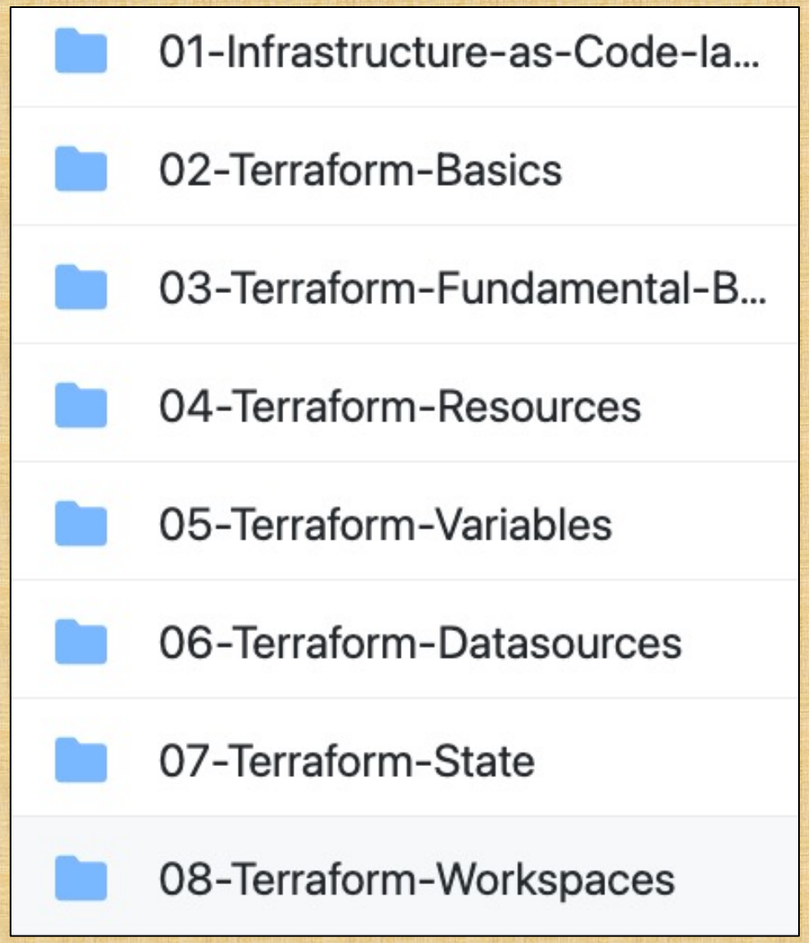
GitHub Step-by-Step Documentation

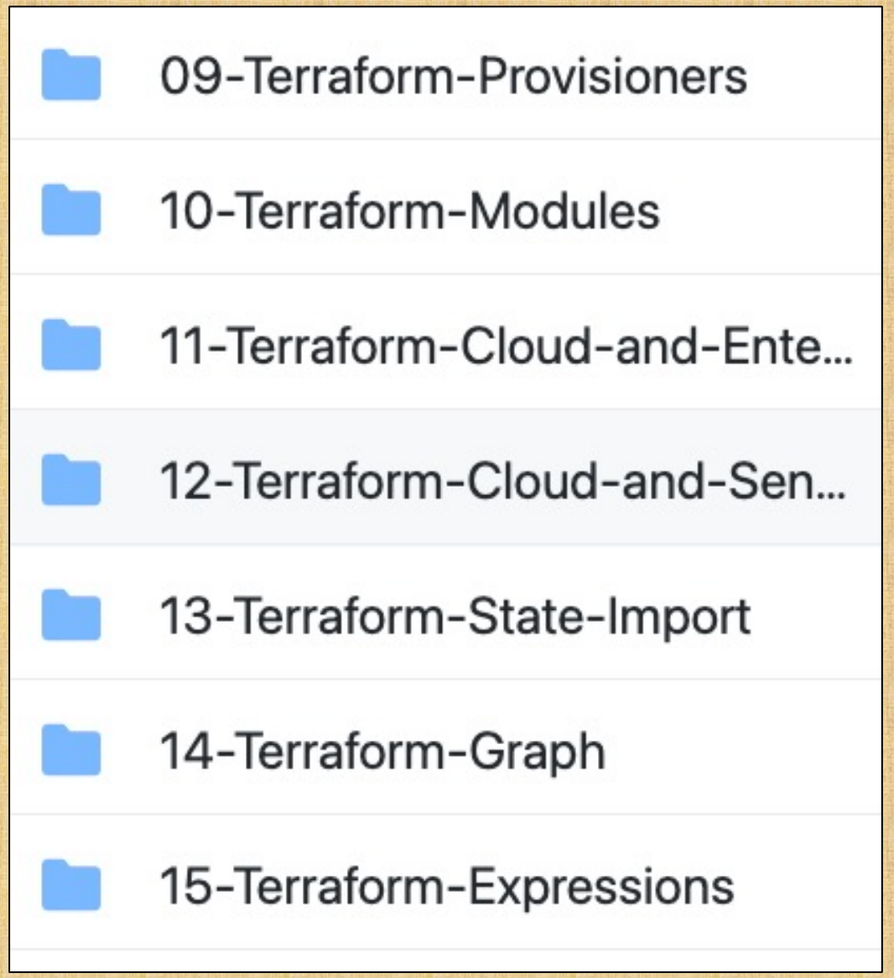
Author: Nho Luong

Skill: DevOps Engineer Lead

Github Step-by-Step Documentation with Practical Examples for each Concept







Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Providers Used

- AWS EC2 Instances
- AWS VPC
- AWS Security Groups
- AWS S3 Buckets
- AWS IAM Users

Core focus will be on mastering Terraform Concepts with Sample Demo’s

Author: Nho Luong

Skill: DevOps Engineer Lead

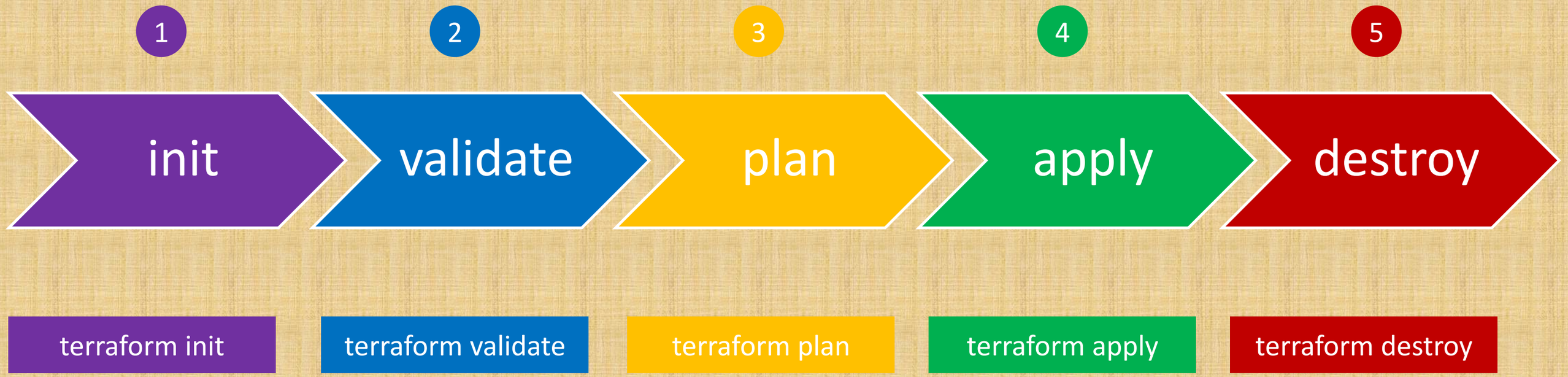
GitHub Repositories

Repository Used For	Repository URL
Course Main Repository with step-by-step documentation	https://github.com/nholuongut/terraform-associate
Terraform Cloud Demo	https://github.com/nholuongut/terraform-cloud-demo1
Terraform Private Module Registry Demo	https://github.com/nholuongut/terraform-aws-s3-website
Terraform Sentinel Policies Demo	https://github.com/nholuongut/terraform-sentinel-policies

Author: Nho Luong

Skill: DevOps Engineer Lead

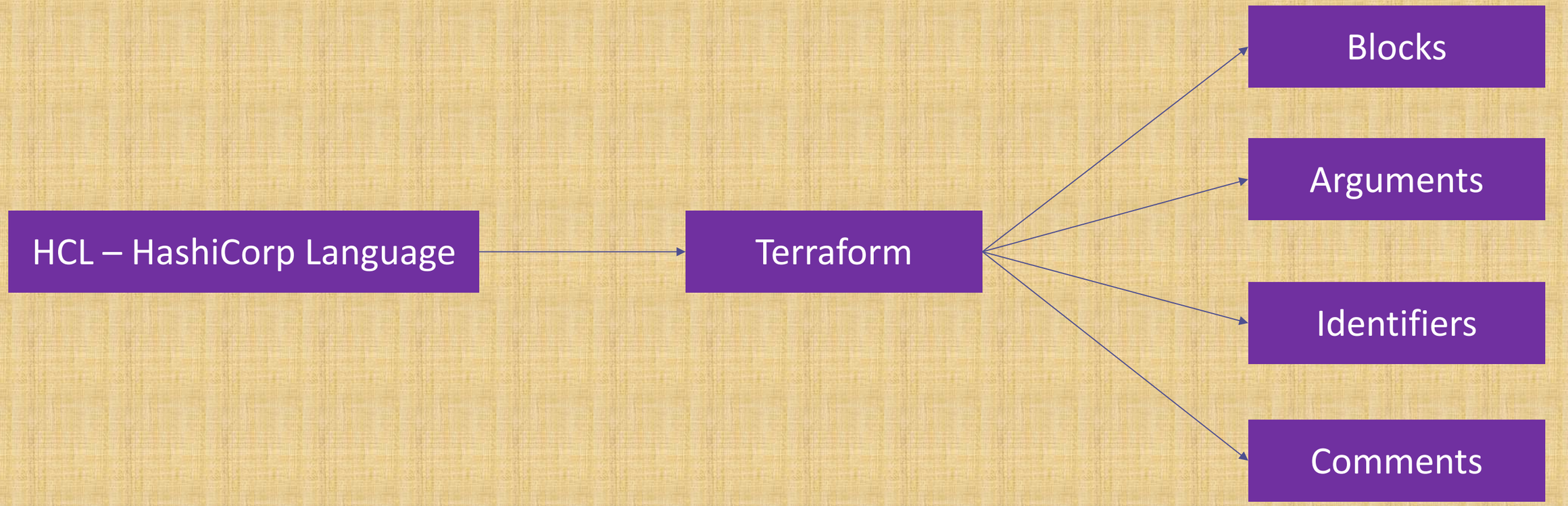
Terraform Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform language uses a limited number of **top-level block** types, which are **blocks** that can appear outside of any other **block** in a TF configuration file.

Terraform Top-Level Blocks

Most of **Terraform's features** are implemented as **top-level** blocks.

Fundamental Blocks

- Terraform Block
- Providers Block
- Resources Block

Variable Blocks

- Input Variables Block
- Output Values Block
- Local Values Block

Calling / Referencing Blocks

- Data Sources Block
- Modules Block

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Providers

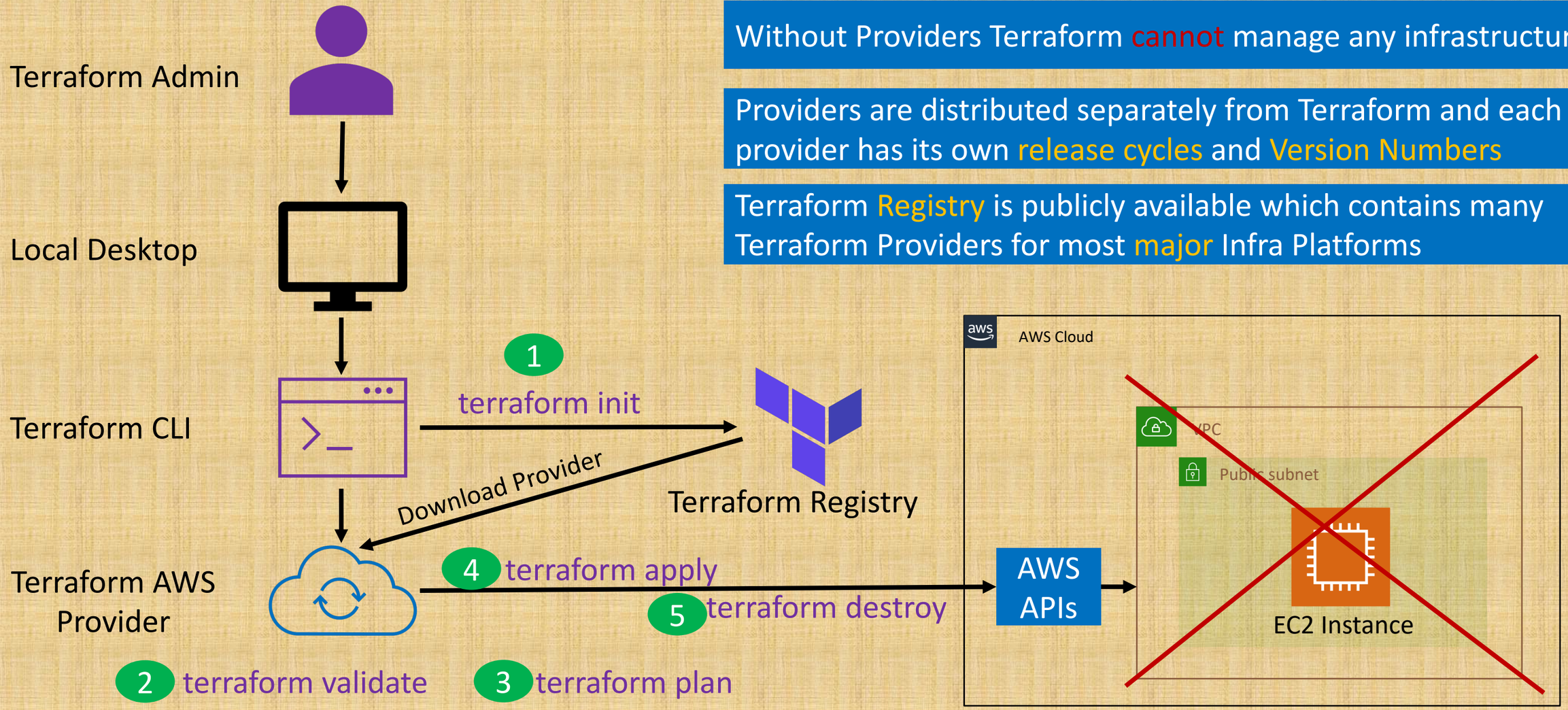
Providers are HEART of Terraform

Every Resource Type (example: EC2 Instance), is implemented by a Provider

Without Providers Terraform cannot manage any infrastructure.

Providers are distributed separately from Terraform and each provider has its own release cycles and Version Numbers

Terraform Registry is publicly available which contains many Terraform Providers for most major Infra Platforms



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Language Basics

Terraform
Resource Syntax

Terraform
Resource Behavior

Terraform
State

Terraform
Resources

Resource
Meta-Argument
count

Resource
Meta-Argument
depends_on

Resource
Meta-Argument
for_each

Resource
Meta-Argument
lifecycle

Author: Nho Luong

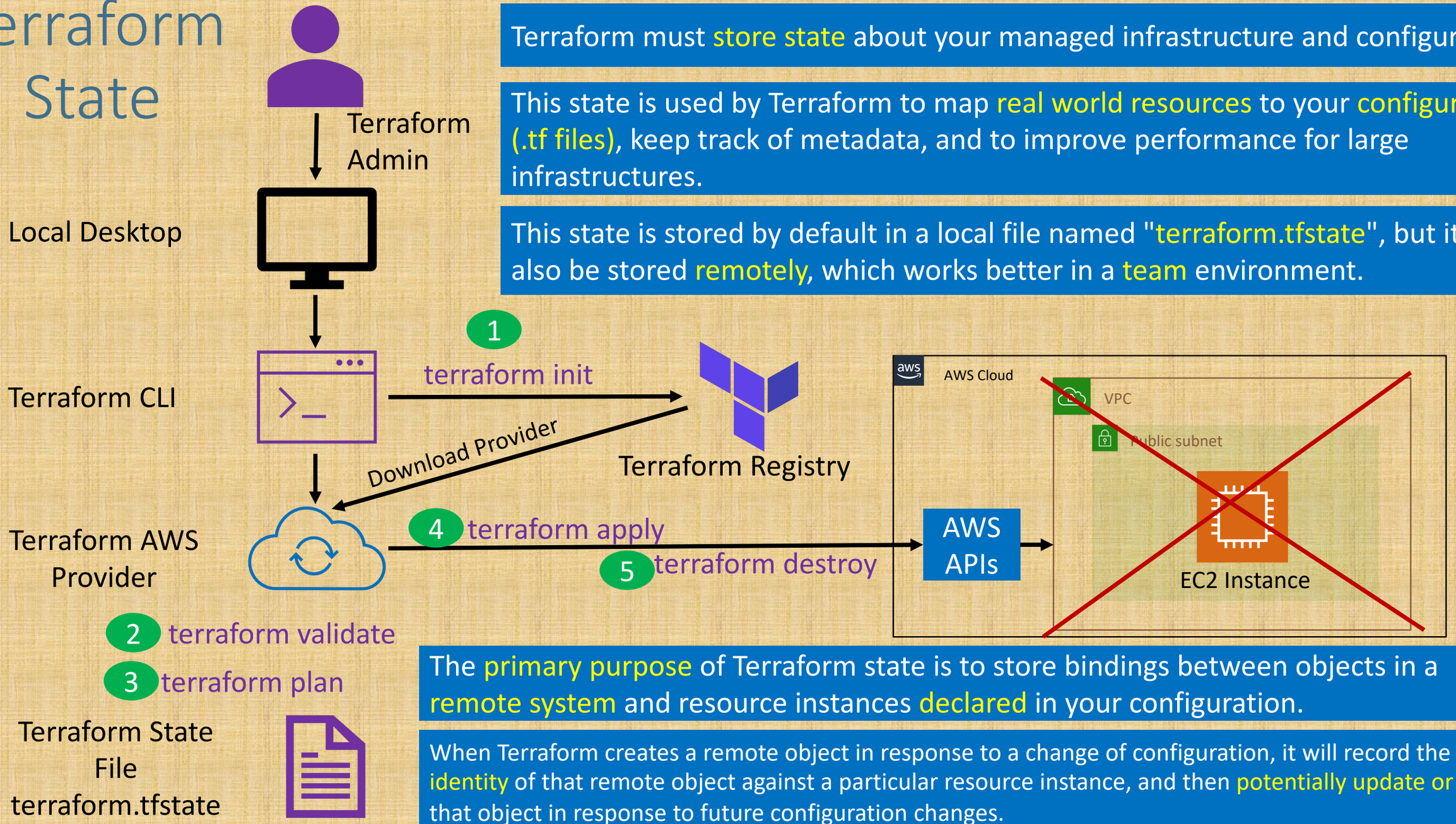
Skill: DevOps Engineer Lead

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 (.tf files), keep track of metadata, and to improve performance for large infrastructures.

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



The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.

When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

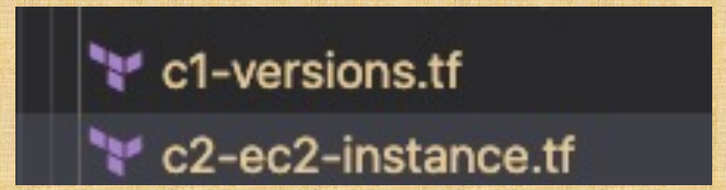
Author: Nho Luong

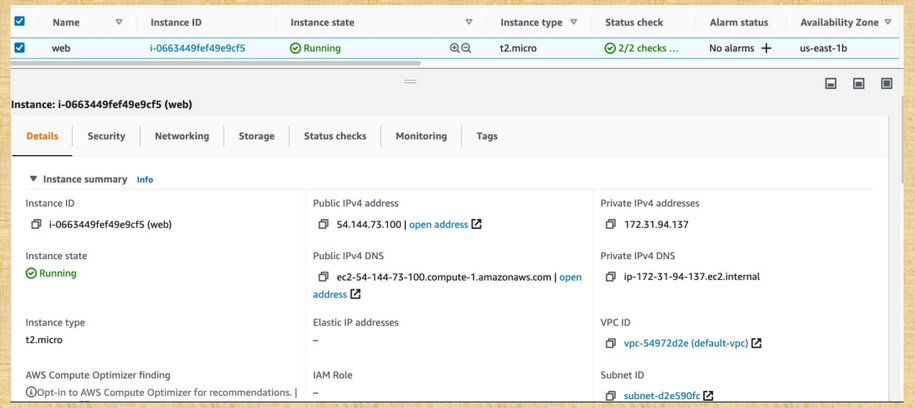
Skill: DevOps Engineer Lead

Desired & Current Terraform States

Terraform Configuration Files

Real World Resource – EC2 Instance





Desired State

Current State

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables

Terraform Input Variables

Terraform Output Values

Terraform Local Values

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Terraform Input Variables

1. Input Variables - Basics
2. Provide Input Variables when prompted during terraform plan or apply
3. Override default variable values using CLI argument -var
4. Override default variable values using Environment Variables (TF_var_aa)
5. Provide Input Variables using terraform.tfvars files
6. Provide Input Variables using <any-name>.tfvars file with CLI argument -var-file
7. Provide Input Variables using auto.tfvars files
8. Implement complex type constructors like List & Map in Input Variables
9. Implement Custom Validation Rules in Variables
10. Protect Sensitive Input Variables

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform State

Terraform Remote State Storage

Terraform Commands from State Perspective

Author: Nho Luong

Skill: DevOps Engineer Lead

What is Terraform Backend ?

Backends are responsible for storing state and providing an API for state locking.

Terraform
State Storage

AWS S3 Bucket

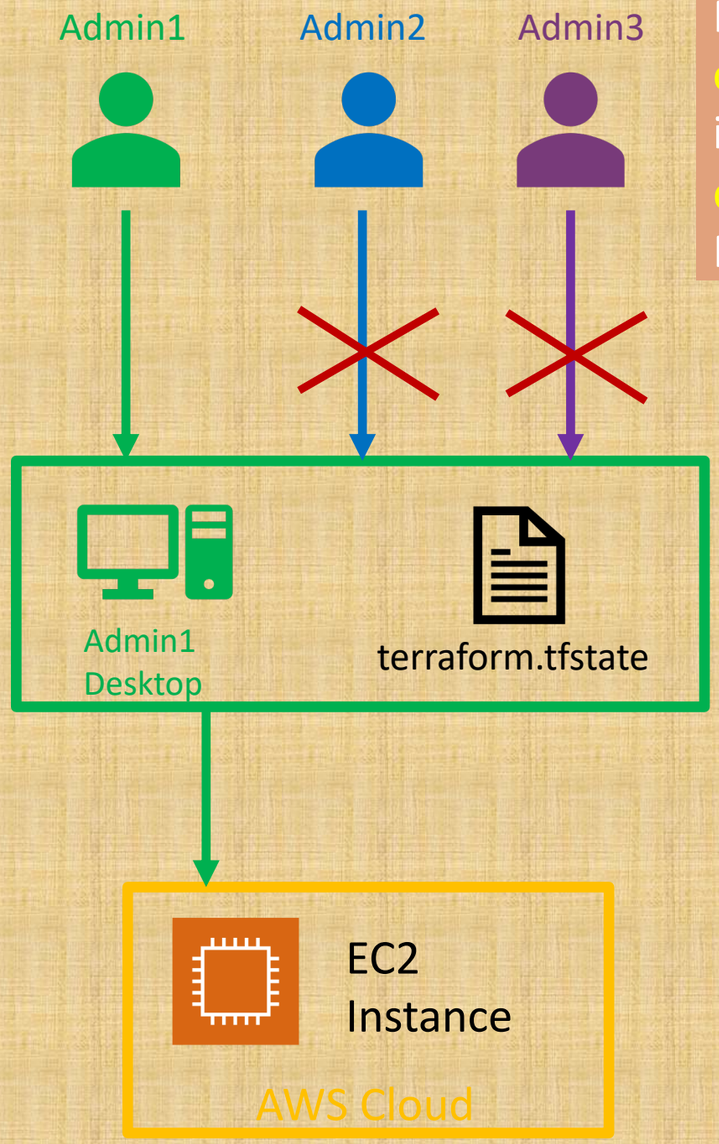
Terraform
State Locking

AWS DynamoDB

Author: Nho Luong

Skill: DevOps Engineer Lead

Local State File



Multiple Team members cannot update the infrastructure as they don't have access to State File

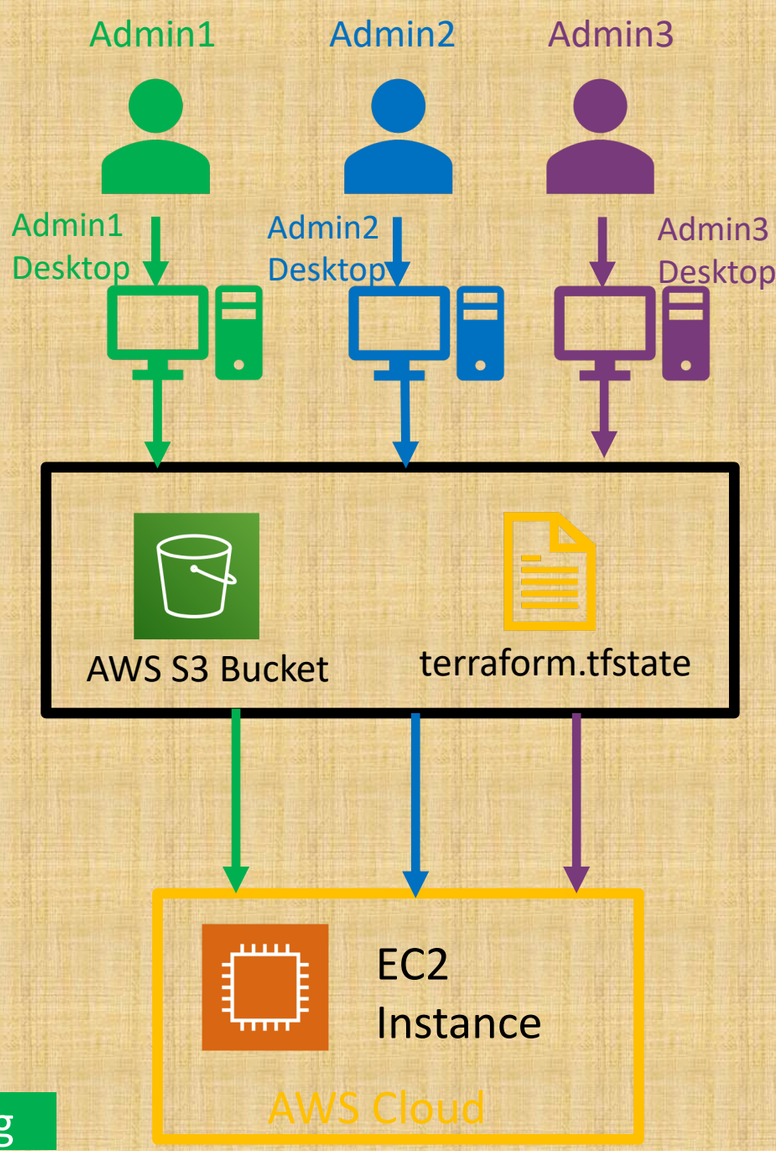
This means we need store the state file in a shared location.

Remote State File

Using Terraform Backend concept we can use AWS S3 as the shared storage for State Files

If two team members are running Terraform at the same time, you may run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.

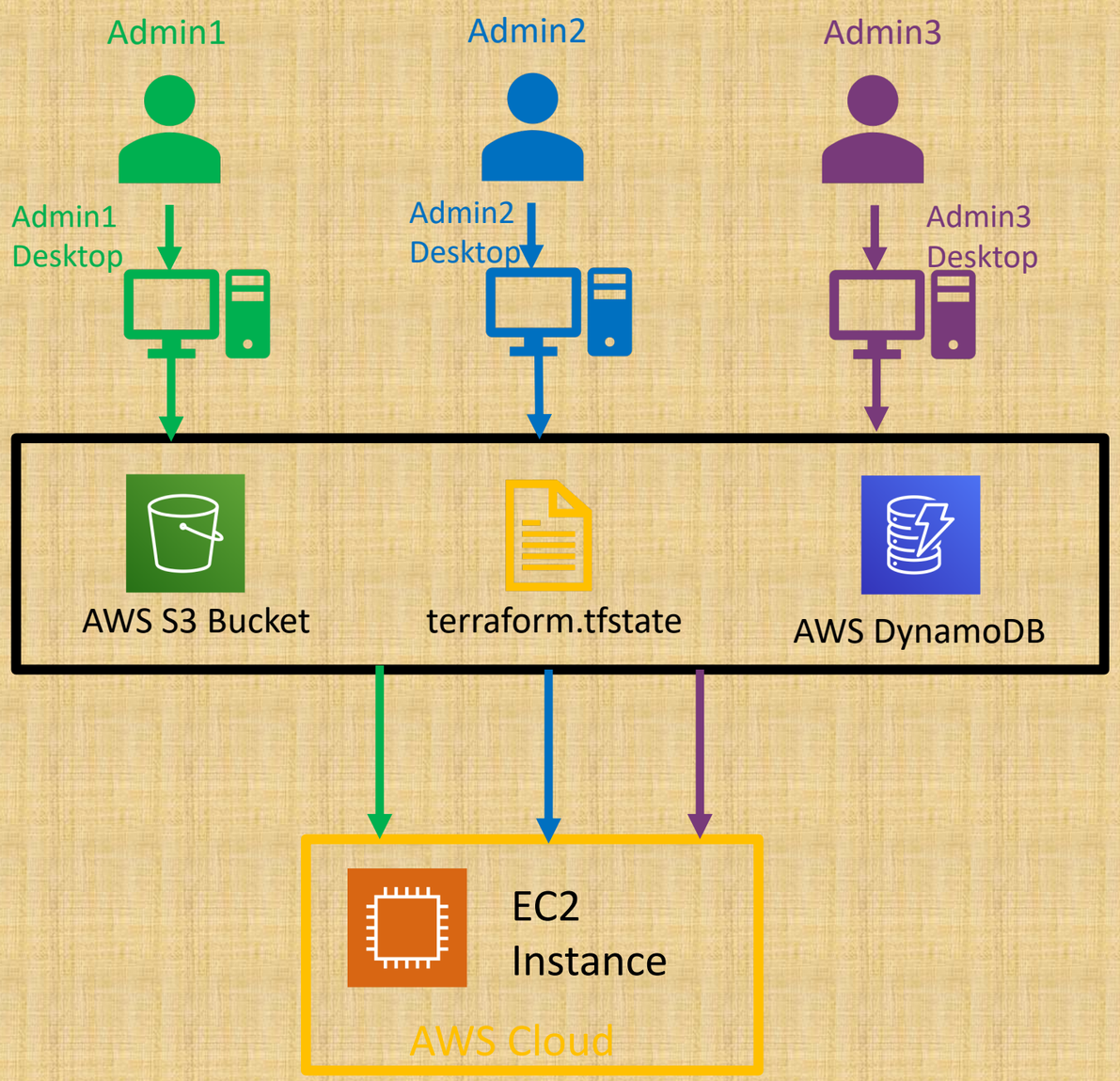
Implement State Locking



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Remote State File with State Locking



Not all backends support State Locking. AWS S3 supports State Locking

State locking happens automatically on all operations that could write state.

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.

If acquiring the lock is taking longer than expected, Terraform will output a status message.

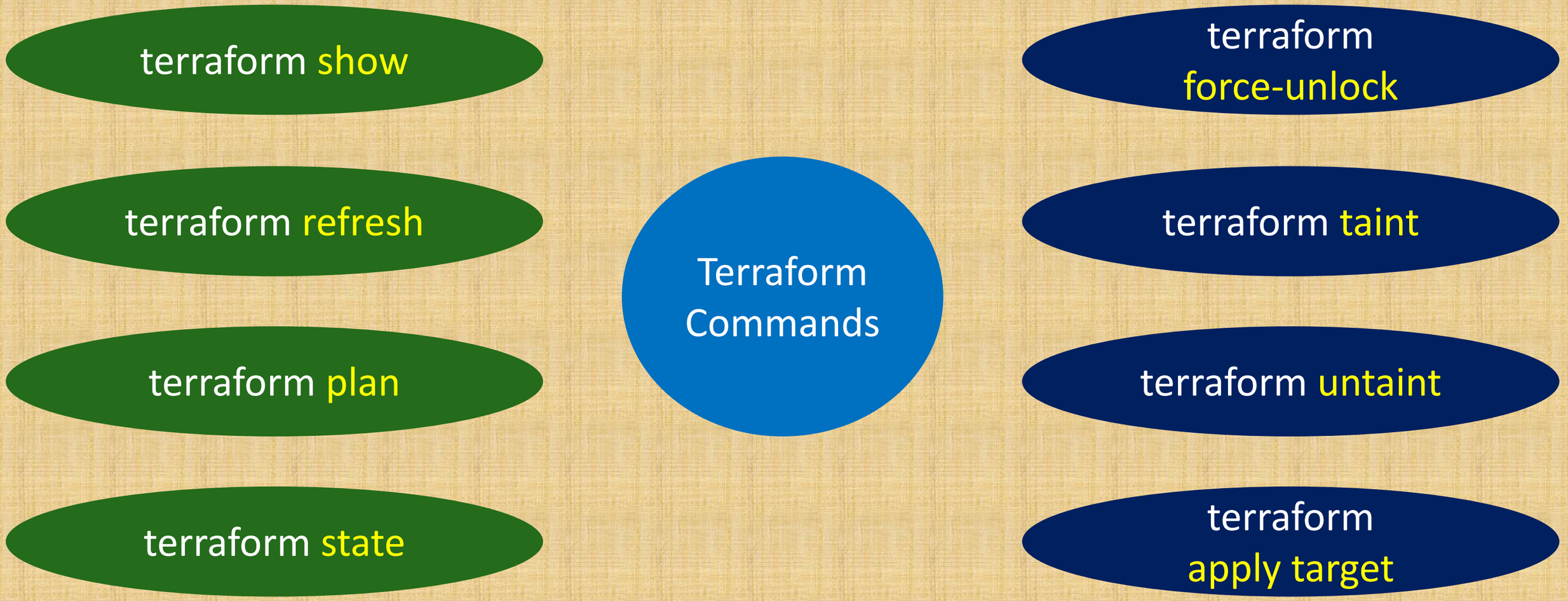
If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Author: Nho Luong

Skill: DevOps Engineer Lead

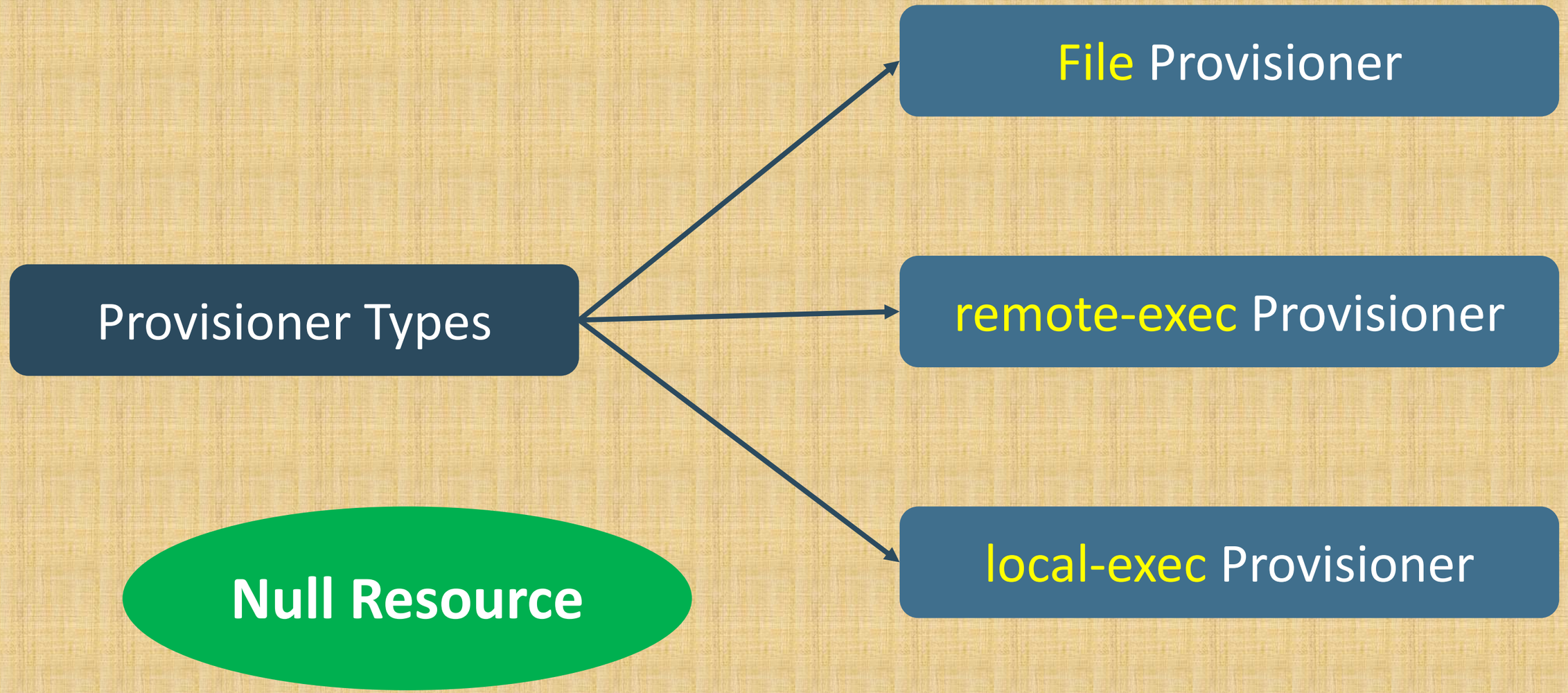
Terraform Commands – State Perspective



Author: Nho Luong

Skill: DevOps Engineer Lead

Types of Provisioners



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Modules

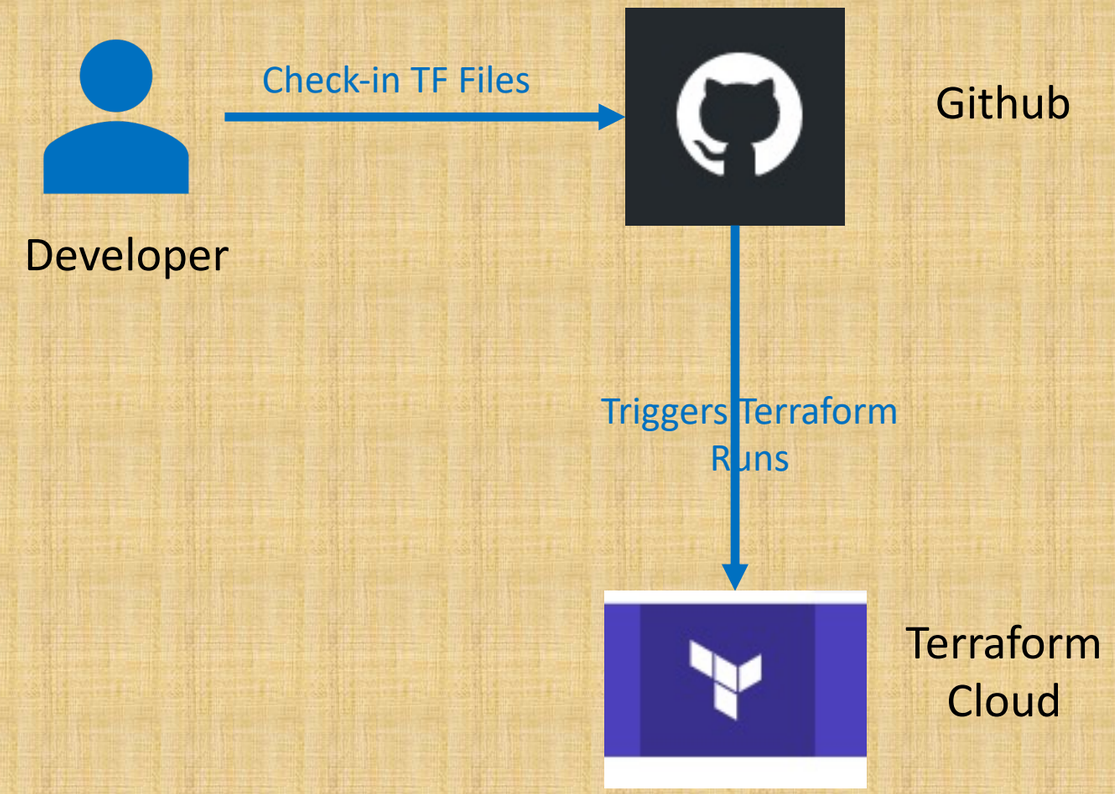
Implement by calling a Public Module from Terraform Registry

Build a Terraform Local Module

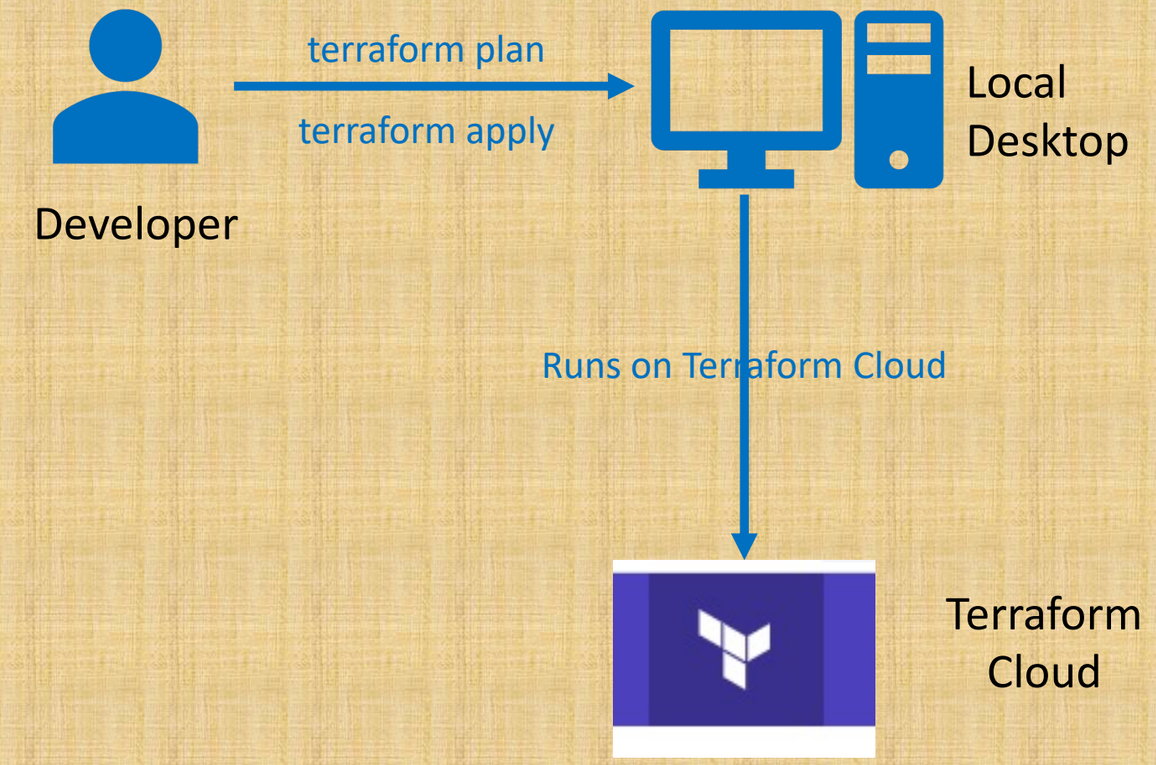
Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform VCS-Driven Workflow



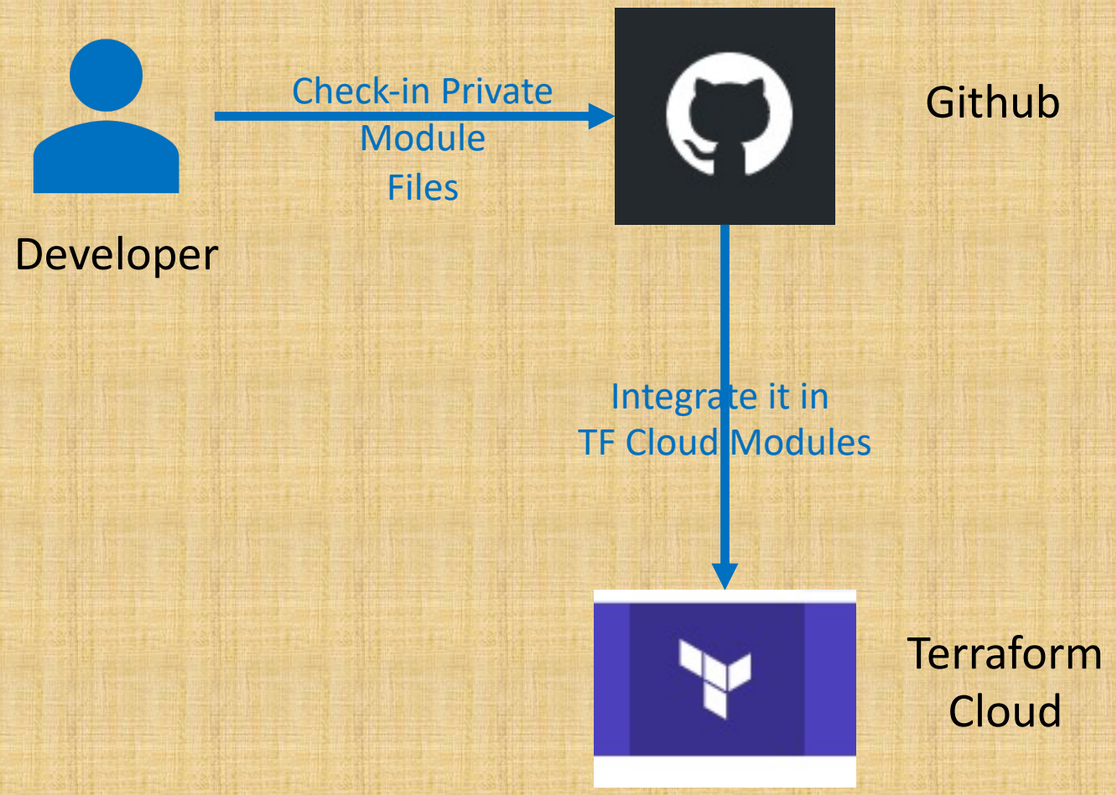
Terraform CLI-Driven Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

Publish Private Module Registry in Terraform Cloud



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud & Sentinel

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Expressions

Expressions are used to refer to or compute values within a configuration

You can experiment with the behavior of Terraform's expressions from the Terraform expression console, by running the terraform console command.

Terraform Functions

Terraform Dynamic Expressions

Terraform Dynamic Blocks

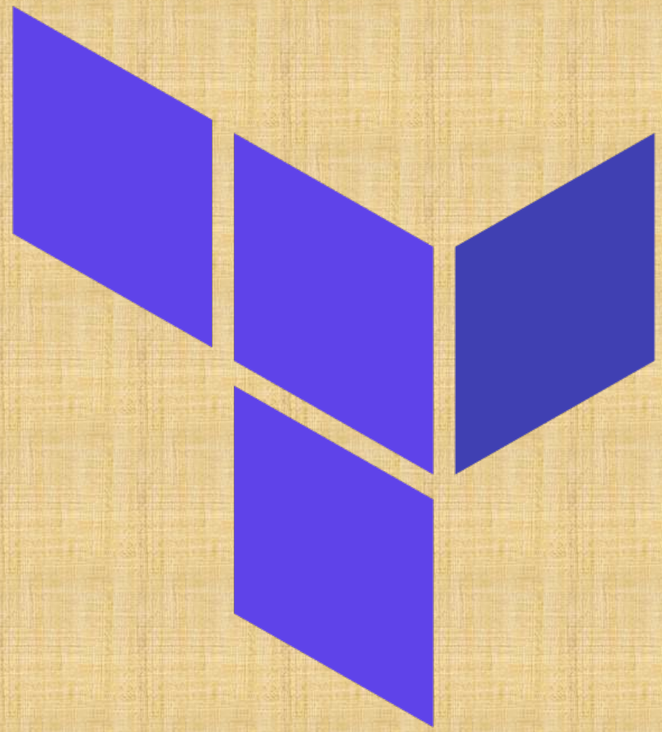
Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Infrastructure as Code

IaC



Author: Nho Luong

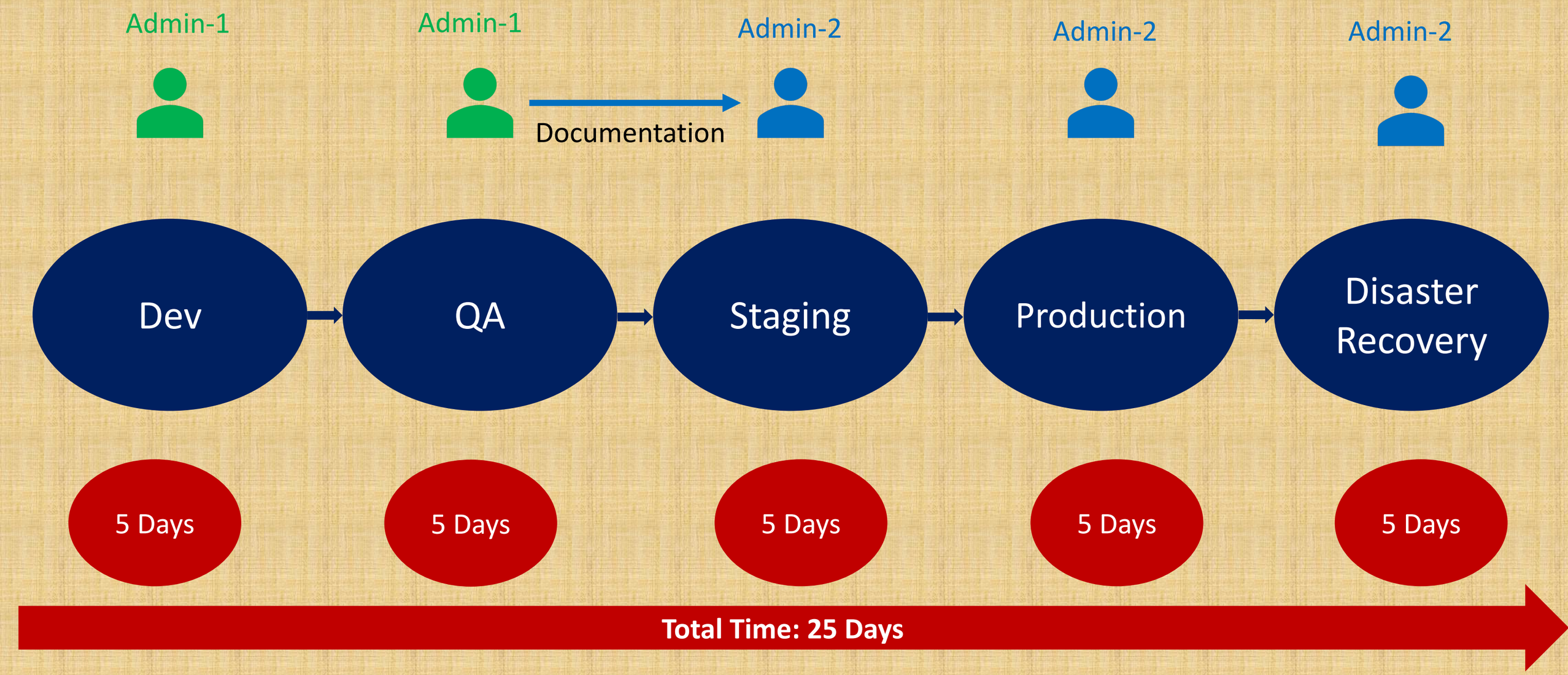
Skill: DevOps Engineer Lead

What is Infrastructure as Code ?

Author: Nho Luong

Skill: DevOps Engineer Lead

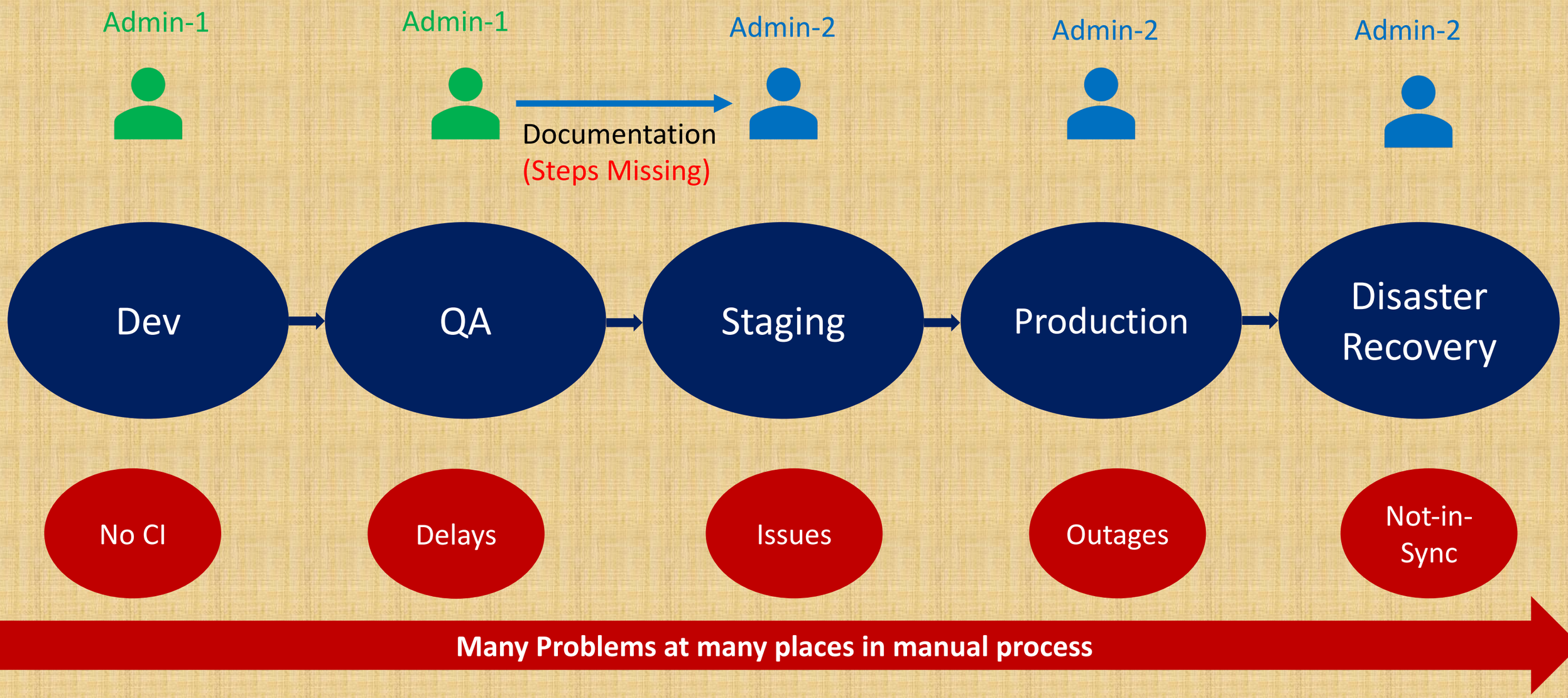
Traditional Way of Managing Infrastructure



Author: Nho Luong

Skill: DevOps Engineer Lead

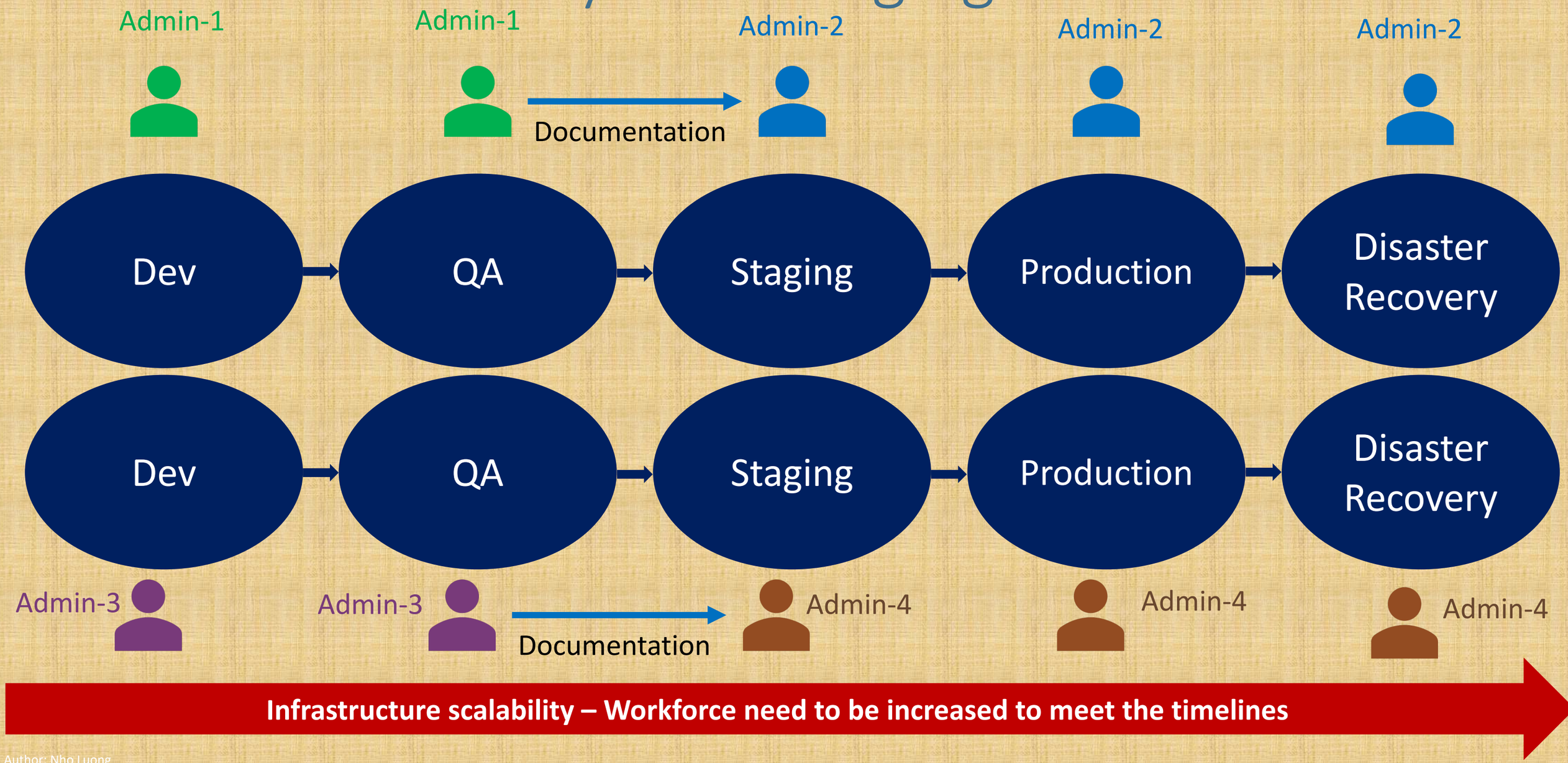
Traditional Way of Managing Infrastructure



Author: Nho Luong

Skill: DevOps Engineer Lead

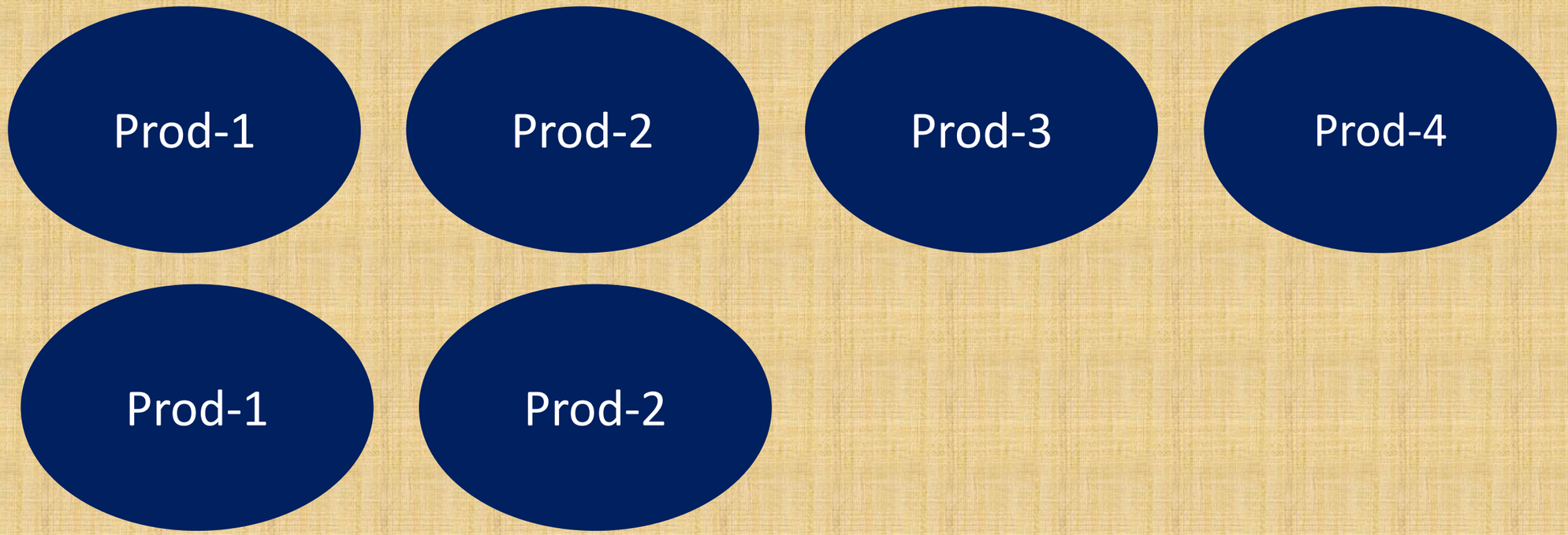
Traditional Way of Managing Infrastructure



Author: Nho Luong

Skill: DevOps Engineer Lead

Traditional Way of Managing Infrastructure



Scale Up

Scale Down

On-Demand Scale-Up and Scale-Down is not an option

Author: Nho Luong

Skill: DevOps Engineer Lead

5 Days

Manage using IaC with Terraform

Admin-1

Check-In TF Code

Github

Triggers TF Runs

Terraform Cloud

DevOps / CI CD for IaC

Scale-Up and Scale-Down On-Demand

Creates Infra

Dev → QA → Staging → Production → Disaster Recovery

One-Time Work

Re-Use Templates

Quick & Fast

Reliable

Tracked for Audit

Total Time: 25 Days reduced to 5 days, Provisioning environments will be in minutes or seconds

Author: Nho Luong

Skill: DevOps Engineer Lead

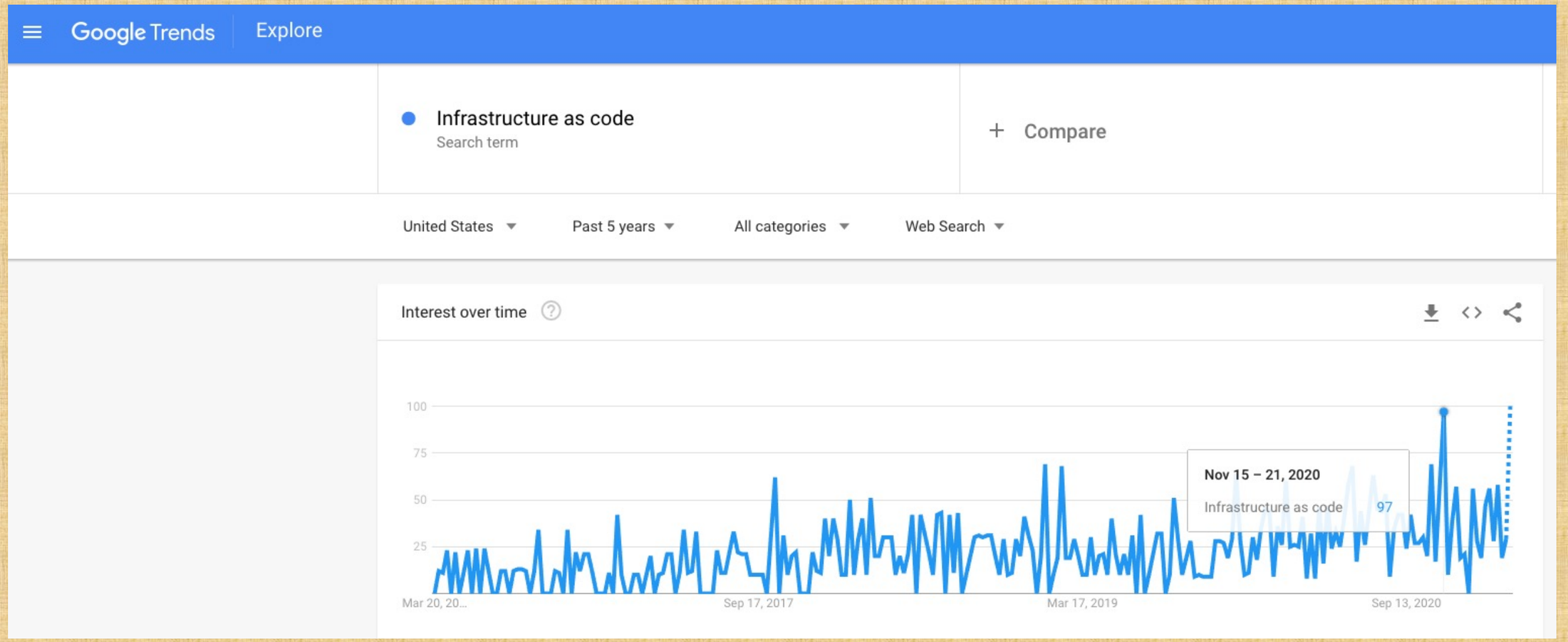
Manage using IaC with Terraform

Visibility	IaC serves as a very clear reference of what resources we created, and what their settings are. We don't have to navigate to the web console to check the parameters.
Stability	If you accidentally change the wrong setting or delete the wrong resource in the web console you can break things. IaC helps solve this, especially when it is combined with version control, such as Git.
Scalability	With IaC we can write it once and then reuse it many times. This means that one well written template can be used as the basis for multiple services, in multiple regions around the world, making it much easier to horizontally scale.
Security	Once again IaC gives you a unified template for how to deploy our architecture. If we create one well secured architecture we can reuse it multiple times, and know that each deployed version is following the same settings.
Audit	Terraform not only creates resources it also maintains the record of what is created in real world cloud environments using its State files.

Author: Nho Luong

Skill: DevOps Engineer Lead

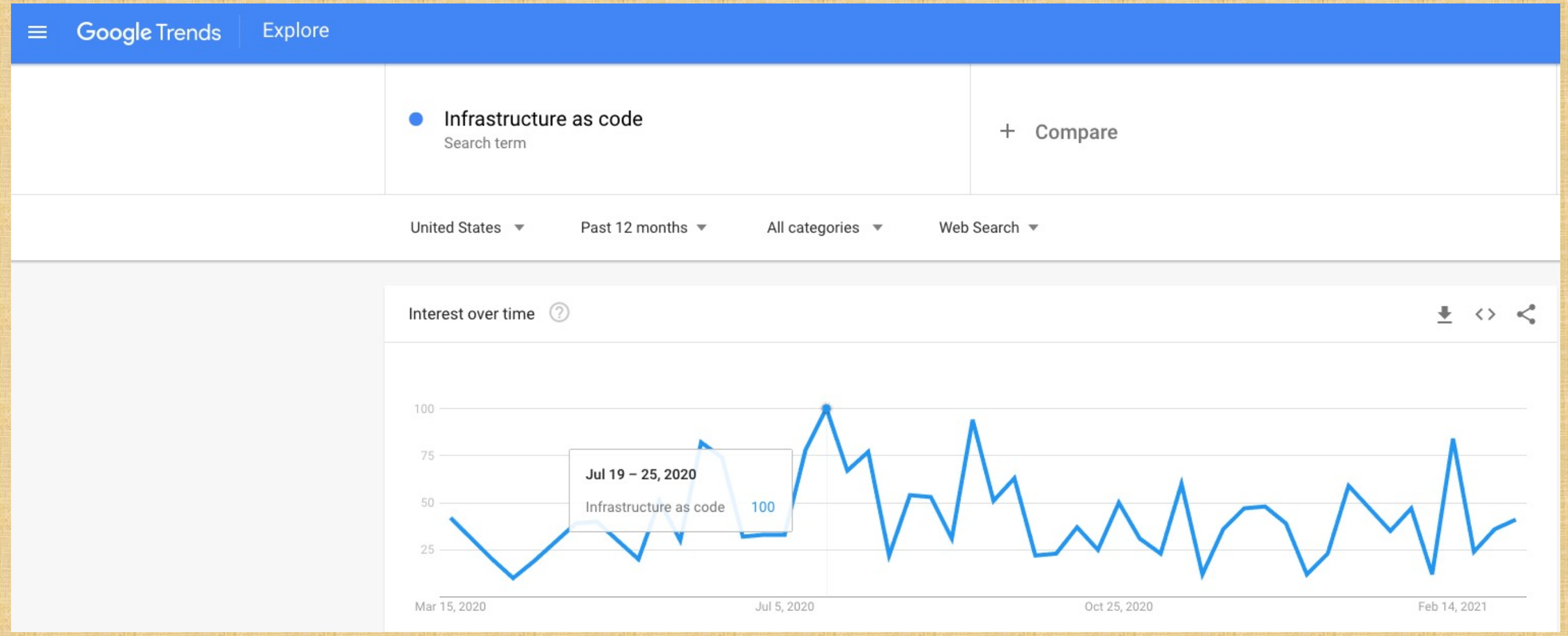
Google Trends – Past 5 Years



Author: Nho Luong

Skill: DevOps Engineer Lead

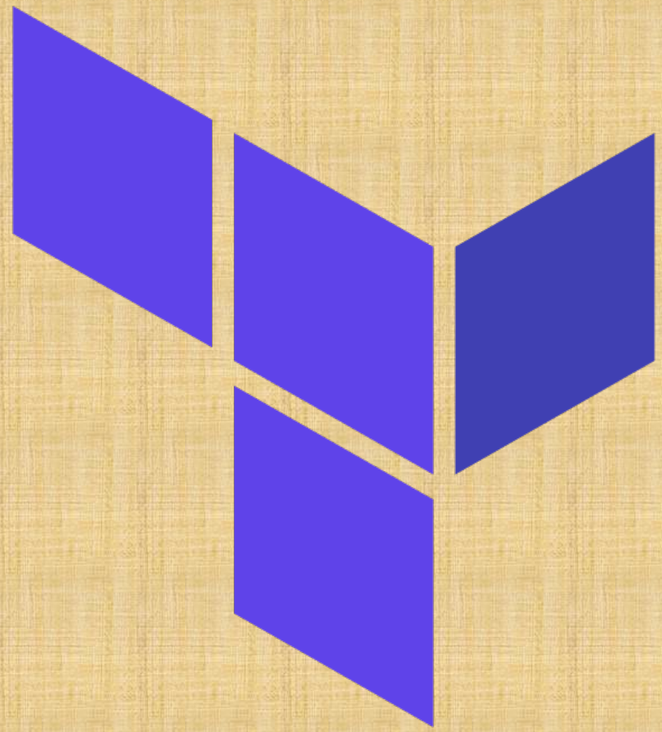
Google Trends – Past 1 Year



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Installation



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Installation

Terraform CLI

AWS CLI

VS Code Editor

Terraform plugin for VS Code

Mac OS

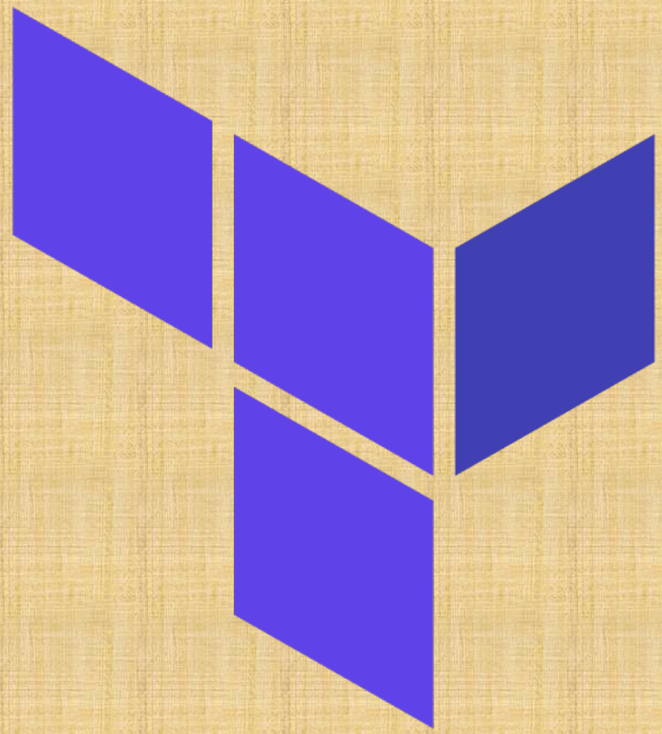
Windows OS

Linux OS

Author: Nho Luong

Skill: DevOps Engineer Lead

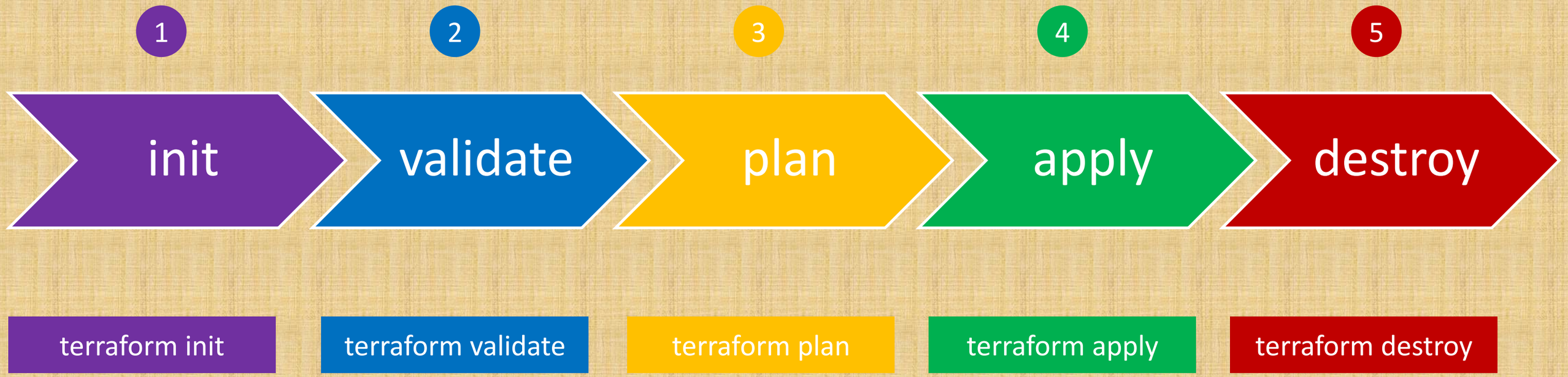
Terraform
Command Basics



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Workflow

1 2 3 4 5

init

- Used to Initialize a working directory containing terraform config files
- This is the first command that should be run after writing a new Terraform configuration
- Downloads Providers

validate

- Validates the terraform configurations files in that respective directory to ensure they are syntactically valid and internally consistent.

plan

- Creates an execution plan
- Terraform performs a refresh and determines what actions are necessary to achieve the desired state specified in configuration files

apply

- Used to apply the changes required to reach the desired state of the configuration.
- By default, apply scan s the current directory for the configuration and applies the changes appropriately.

destroy

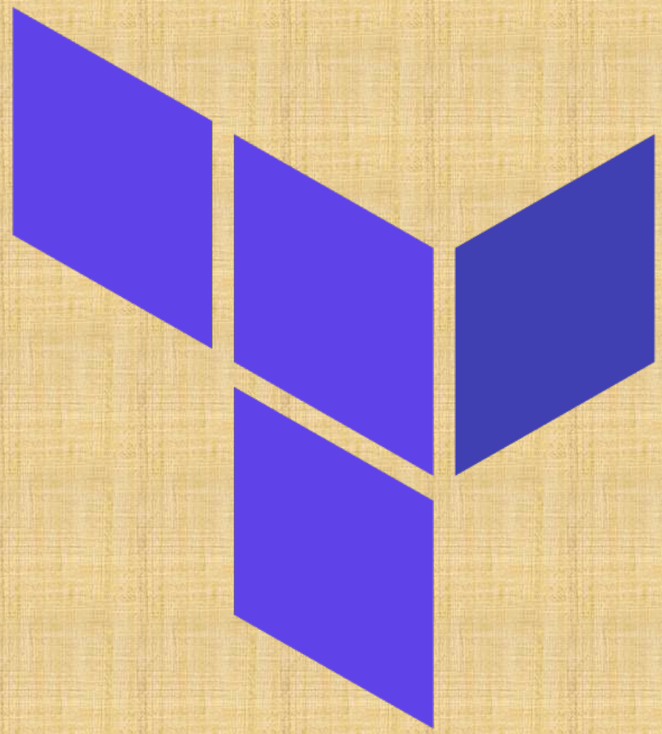
- Used to destroy the Terraform-managed infrastructure
- This will ask for confirmation before destroying.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Language Basics

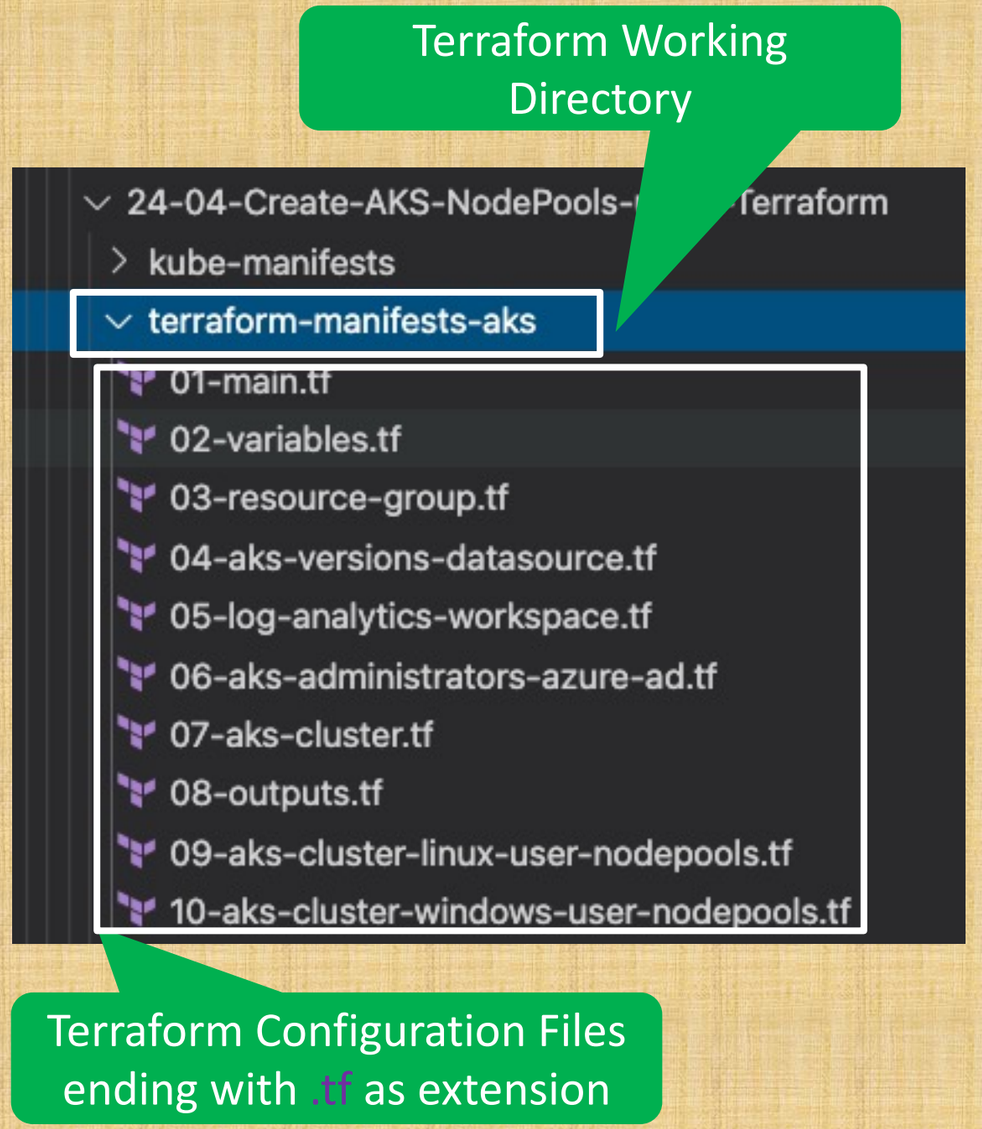


Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Files

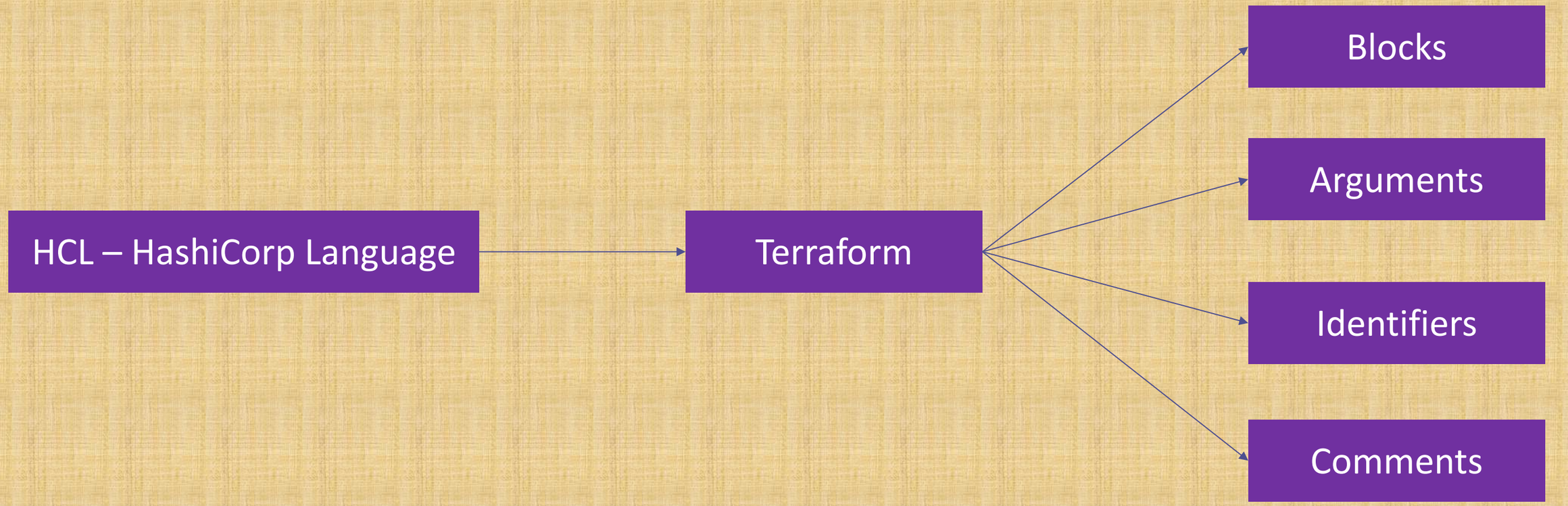
- Code in the Terraform language is stored in plain text files with the .tf file extension.
- There is also a JSON-based variant of the language that is named with the .tf.json file extension.
- We can call the files containing terraform code as Terraform Configuration Files or Terraform Manifests



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>"   {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

# AWS Example
resource "aws_instance" "ec2demo" {
  ami           = "ami-04d29b6f966df1537"
  instance_type = "t2.micro"
}
```

Block Labels

Block Type

Top Level & Block inside Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners, resource specific blocks like tags

Arguments

Based on Block Type block labels will be 1 or 2
Example:
Resource – 2 labels
Variables – 1 label

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>"   {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

# AWS Example
resource "aws_instance" "ec2demo" {
  ami           = "ami-04d29b6f966df1537"
  instance_type = "t2.micro"
}
```

Argument Name [or] Identifier

Argument Value [or] Expression

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

Single Line Comments with # or //

Multi-line comment

```
# EC2 Instance Resource
resource "aws_instance" "ec2demo" {
  ami           = "ami-0885b1f6bd170450c" // Ubuntu 20.04 LTS
  instance_type = "t2.micro"
  /*
  Multi-line comments
  Line-1
  Line-2
  */
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform language uses a limited number of **top-level block** types, which are **blocks** that can appear outside of any other **block** in a TF configuration file.

Terraform Top-Level Blocks

Most of **Terraform's features** are implemented as **top-level** blocks.

Fundamental Blocks

- Terraform Block
- Providers Block
- Resources Block

Variable Blocks

- Input Variables Block
- Output Values Block
- Local Values Block

Calling / Referencing Blocks

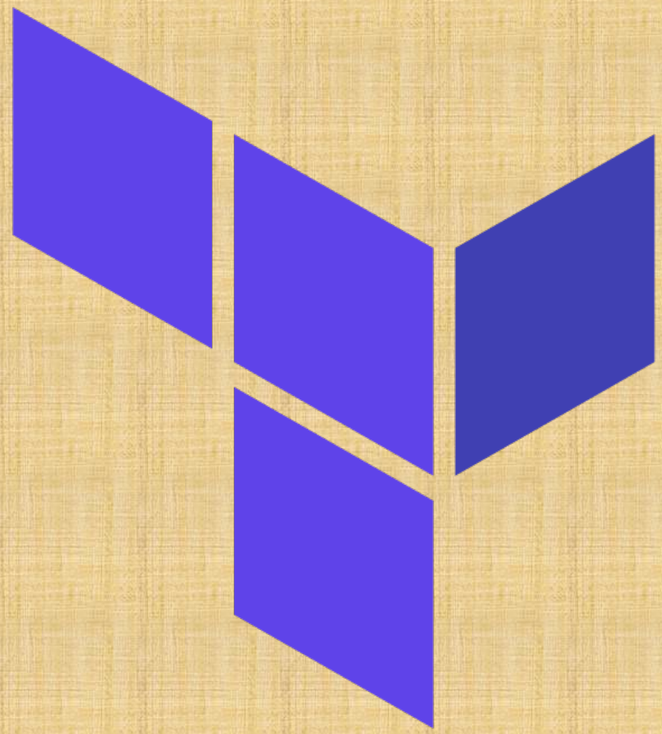
- Data Sources Block
- Modules Block

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Fundamental Blocks



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Basic Blocks

Terraform Block

- Special block used to configure some behaviors
- Specifying a required Terraform Version
- Specifying Provider Requirements
- Configuring a Terraform Backend (Terraform State)

Provider Block

- HEART of Terraform
- Terraform relies on providers to interact with Remote Systems
- Declare providers for Terraform to install providers & use them
- Provider configurations belong to Root Module

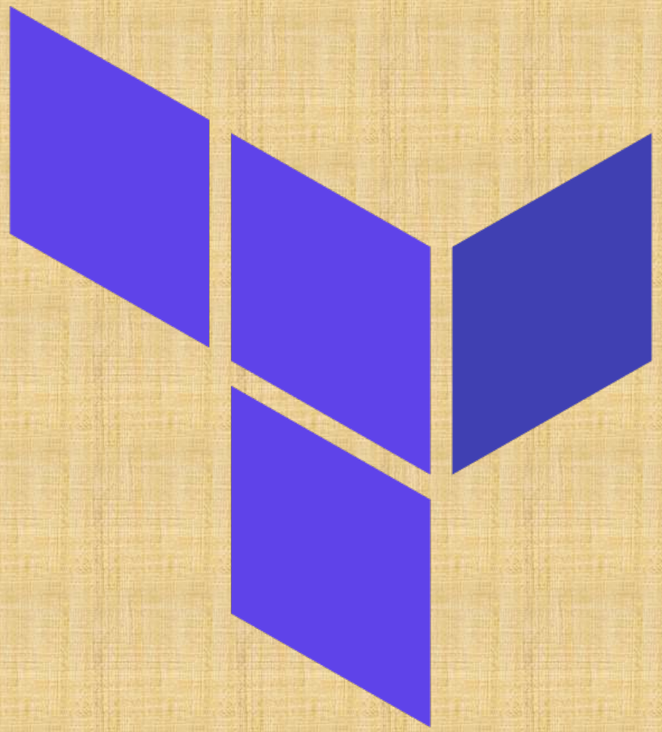
Resource Block

- Each Resource Block describes one or more Infrastructure Objects
- Resource Syntax: How to declare Resources?
- Resource Behavior: How Terraform handles resource declarations?
- Provisioners: We can configure Resource post-creation actions

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Block



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Block

- This block can be called in 3 ways. All means the same.
  - Terraform Block
  - Terraform Settings Block
  - Terraform Configuration Block
- Each terraform block can contain a number of settings related to Terraform's behavior.
- Within a terraform block, only constant values can be used; arguments may not refer to named objects such as resources, input variables, etc, and may not use any of the Terraform language built-in functions.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Block from 0.13 onwards

Terraform 0.12 and earlier:

```
# Configure the AWS Provider
provider "aws" {
  version = "~> 3.0"
  region  = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

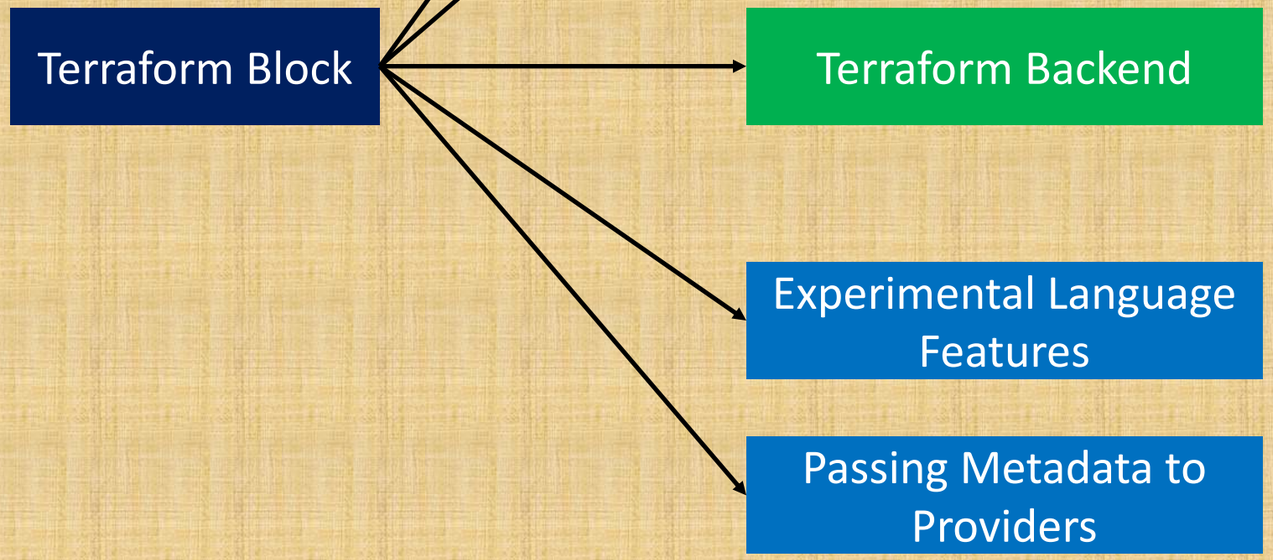
Terraform 0.13 and later:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Block

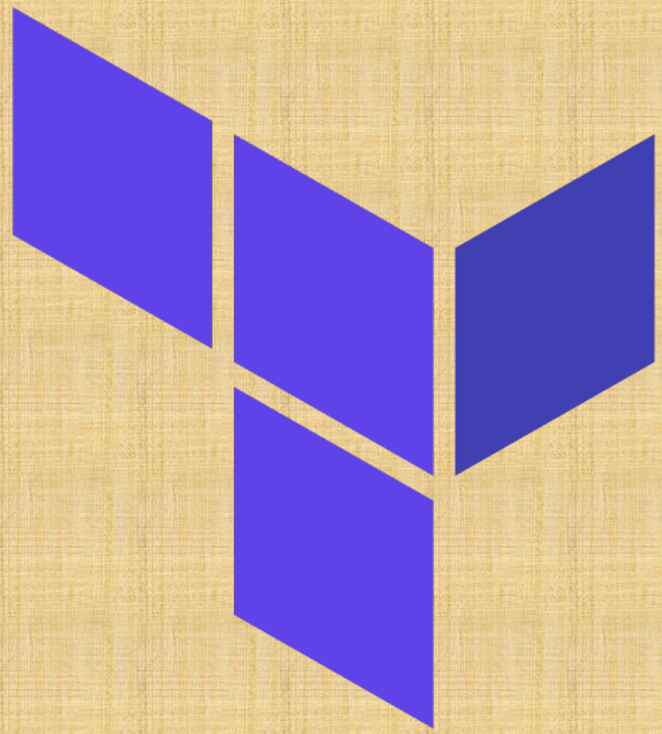



```hcl
terraform {
  # Required Terraform Version
  required_version = "~> 0.14.3"
  # Required Providers and their Versions
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.21" # Optional but recommended
    }
  }
  # Remote Backend for storing Terraform State in S3 bucket
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "us-east-1"
  }
  # Experimental Features (Not required)
  experiments = [ example ]
  # Passing Metadata to Providers (Super Advanced - Terrafor
  provider_meta "my-provider" {
    hello = "world"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Providers



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Providers

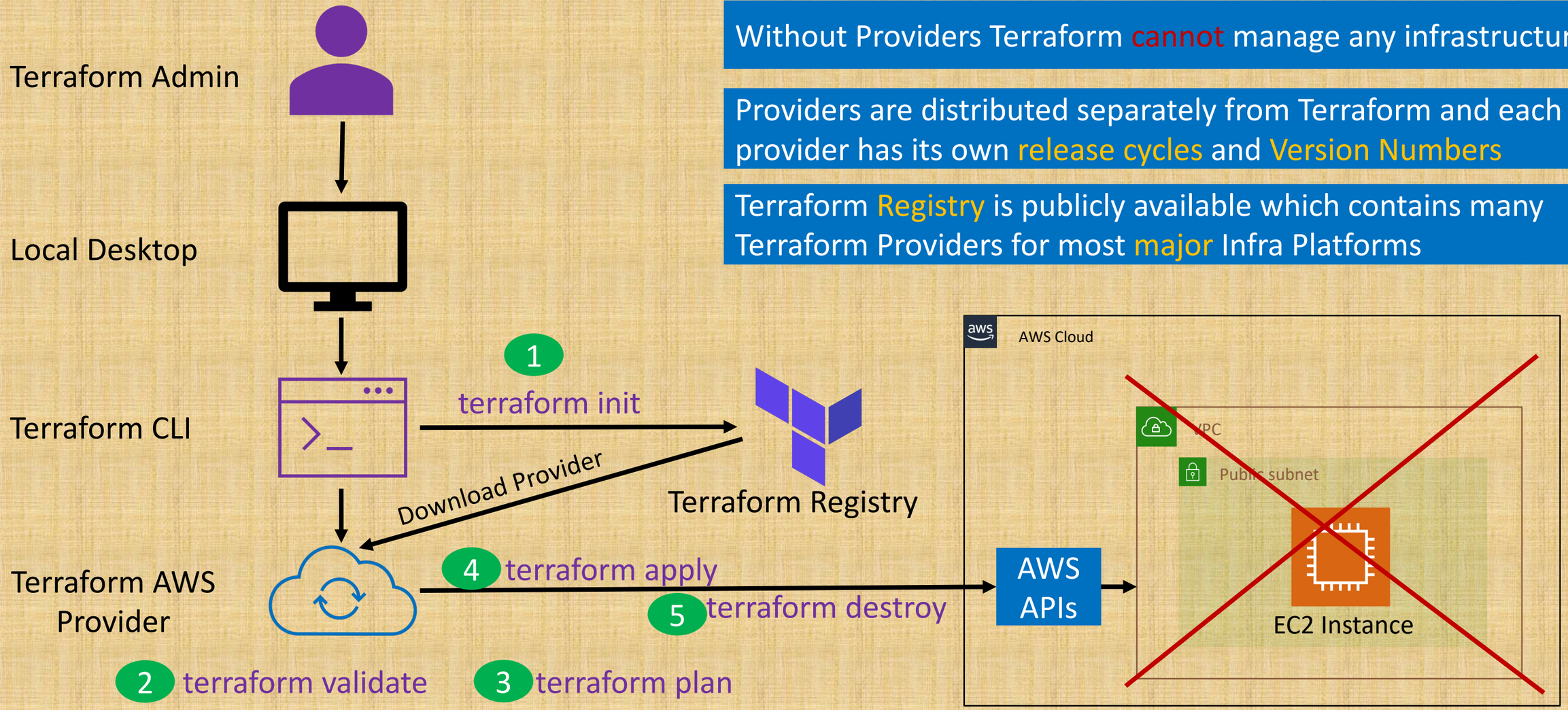
Providers are HEART of Terraform

Every Resource Type (example: EC2 Instance), is implemented by a Provider

Without Providers Terraform cannot manage any infrastructure.

Providers are distributed separately from Terraform and each provider has its own release cycles and Version Numbers

Terraform Registry is publicly available which contains many Terraform Providers for most major Infra Platforms



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Providers

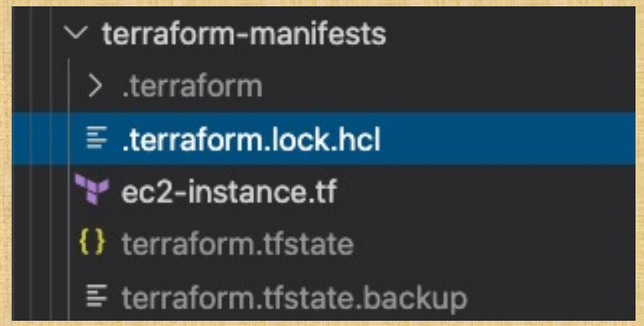
Provider Requirements

```
# Terraform Block
terraform {
  required_version = "~> 0.14.3"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}
```

Provider Configuration

```
# Provider Block
provider "aws" {
  profile = "default"
  region  = "us-east-1"
}
```

Dependency Lock File




```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version = "3.22.0"
  hashes = [
    "h1:f/Tz8zv1Zb78ZaiyJkQ0MGIViZwbYrLuQk3kojPM91c=",
    "zh:4a9a66caf1964cdd3b61fb3ebb0da417195a5529cb8e496f266b0778335d11c8",
    "zh:514f2f006ae68db715d86781673faf9483292deab235c7402ff306e0e92ea11a",
    "zh:5277b61109fddb9011728f6650ef01a639a0590aeffe34ed7de7ba10d0c31803",
    "zh:67784dc8c8375ab37103eea1258c3334ee92be6de033c2b37e3a2a65d0005142",
    "zh:76d4c8be2ca4a3294fb51fb58de1fe03361d3bc403820270cc8e71a04c5fa806",
    "zh:8f90b1cfdcf6e8fb1a9d0382ecaa5056a3a84c94e313fbf9e92c89de271cdede",
    "zh:d0ac346519d0df124df89be2d803eb53f373434890f6ee3fb37112802f9eac59",
    "zh:d6256feedada82cbfb3b1dd6dd9ad02048f23120ab50e6146a541cb11a108cc1",
    "zh:db2fe0d2e77c02e9a74e1ed694aa352295a50283f9a1cf896e5be252af14e9f4",
    "zh:eda61e889b579bd90046939a5b40cf5dc9031fb5a819fc3e4667a78bd432bdb2",
  ]
}
```


Author: Nho Luong

Skill: DevOps Engineer Lead

Dependency Lock File

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version = "3.22.0"
  hashes = [
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Required Providers

```
# Terraform Block
terraform {
  required_version = "~> 0.14.3"
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# Provider Block
provider "aws" {
  profile = "default"
  region  = "us-east-1"
}
```

Local Names

Local Names are Module specific and should be unique per-module

Terraform configurations always refer to local name of provider outside required_provider block

Users of a provider can choose any local name for it (myaws, aws1, aws2).

Recommended way of choosing local name is to use preferred local name of that provider (For AWS Provider: hashicorp/aws, preferred local name is aws)

Source

It is the primary location where we can download the Terraform Provider

Source addresses consist of three parts delimited by slashes (/)

[<HOSTNAME>/]<NAMESPACE>/<TYPE>

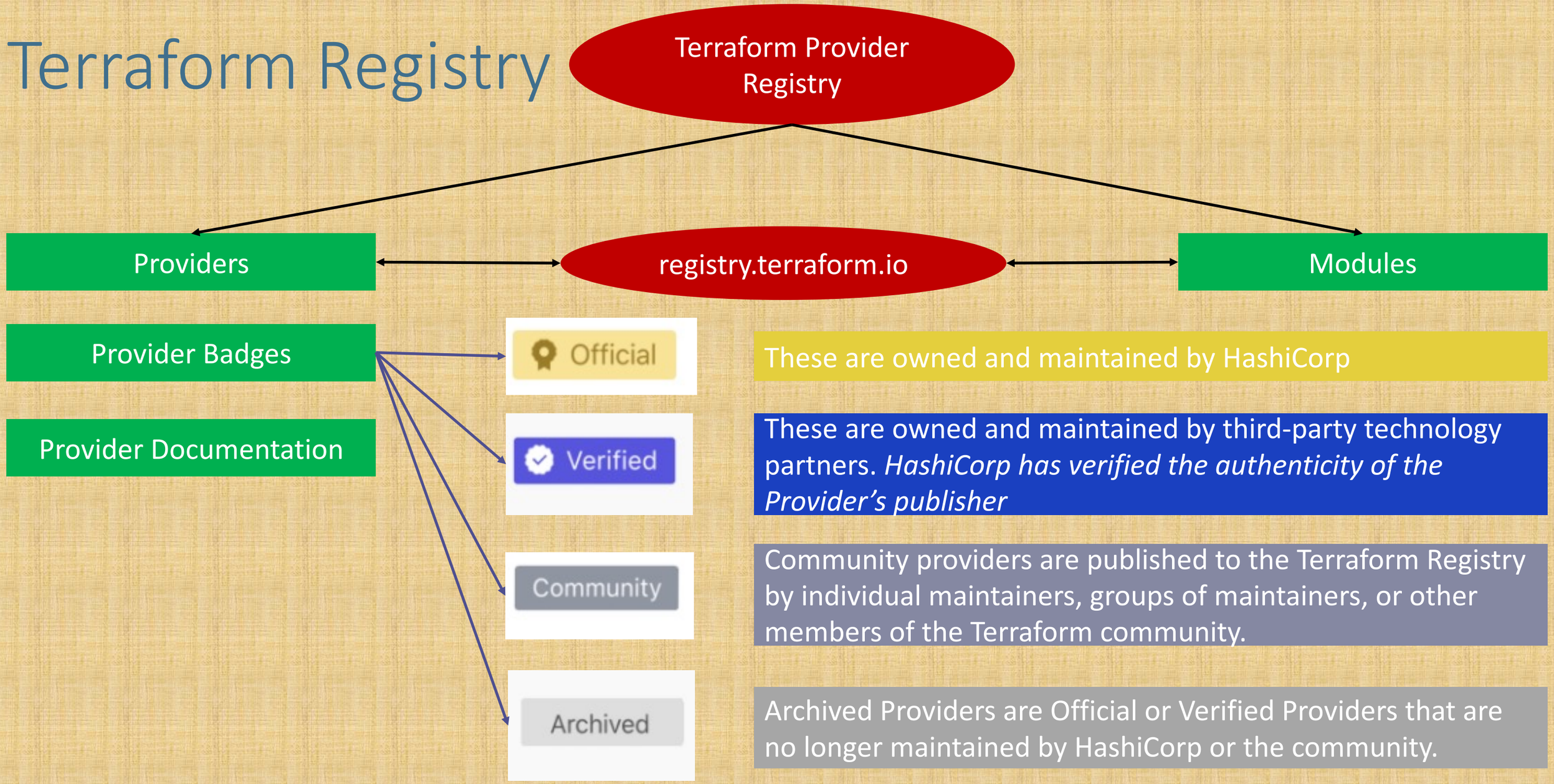
registry.terraform.io/hashicorp/aws

Registry Name is optional as default is going to be Terraform Public Registry

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Registry

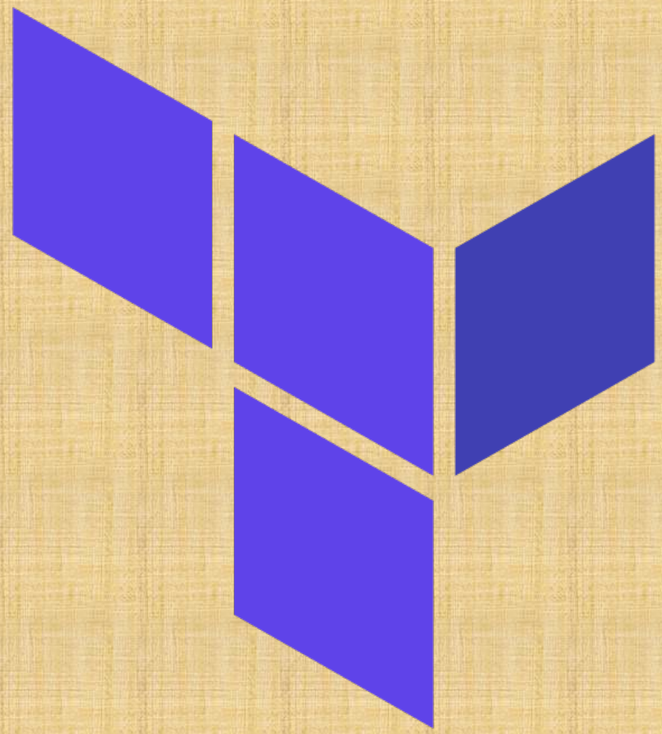


Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Multiple Providers



Author: Nho Luong

Skill: DevOps Engineer Lead

Multiple Providers

We can define multiple configurations for the same provider, and select which one to use on a per-resource or per-module basis.

The primary reason for this is to support multiple regions for a cloud platform

We can use the alternate provider in a resource, data or module by referencing it as <PROVIDER NAME>.<ALIAS>

```
# Provider-1 for us-east-1 (Default Provider)
provider "aws" {
  region = "us-east-1"
  profile = "default"
}

# Provider-2 for us-west-1
provider "aws" {
  region = "us-west-1"
  profile = "default"
  alias = "aws-west-1"
}
```

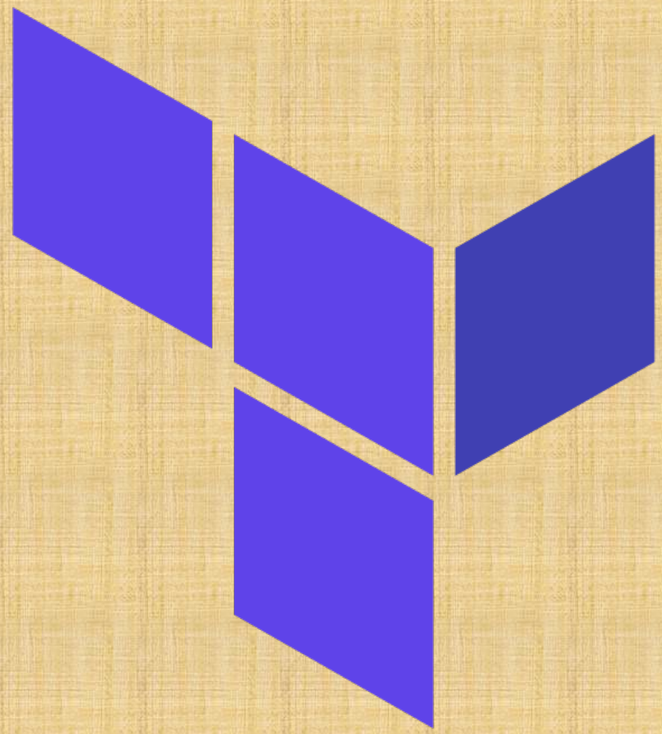
```
# Resource Block to Create VPC in us-west-1
resource "aws_vpc" "vpc-us-west-1" {
  cidr_block = "10.2.0.0/16"
  #<PROVIDER NAME>.<ALIAS>
  provider = aws.aws-west-1
  tags = {
    "Name" = "vpc-us-west-1"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Dependency Lock File



Author: Nho Luong

Skill: DevOps Engineer Lead

Dependency Lock File

Terraform

New feature added from Terraform v0.14 & later

Providers

Terraform configuration refers to two different kinds of external dependency that come from outside of its own codebase

Modules

Version Constraints within the configuration itself determine which versions of dependencies are *potentially* compatible

Dependency Lock File: After selecting a specific version of each dependency using Version Constraints Terraform remembers the decisions it made in a dependency lock file so that it can (by default) make the same decisions again in future.

Location of Lock File: Current Working Directory

Very Important: Lock File currently tracks only Provider Dependencies. For modules continue to use exact version constraint to ensure that Terraform will always select the same module version.

Checksum Verification: Terraform will also verify that each package it installs matches at least one of the checksums it previously recorded in the lock file, if any, returning an error if none of the checksums match

Author: Nho Luong

Skill: DevOps Engineer Lead

Dependency Lock File

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version     = "2.50.0"
  constraints = ">= 2.0.0"
  hashes = [
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Importance of Dependency Lock File

Provider	Version Constraint	`terraform init` (no lock file)	`terraform init` (lock file)
aws	>= 2.0	Latest version (3.18.0)	Lock file version (2.50.0)
random	3.0.0	3.0.0	Lock file version (3.0.0)

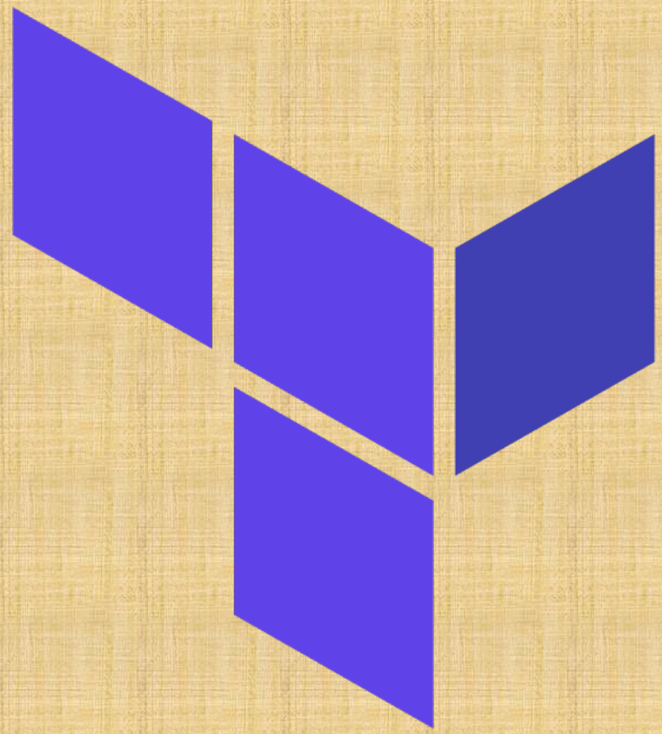
If Terraform did not find a lock file, it would download the latest versions of the providers that fulfill the version constraints you defined in the required_providers block inside Terraform Settings Block.

If we have lock file, the lock file causes Terraform to always install the same provider version, ensuring that runs across your team or remote sessions will be consistent.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Resources
Introduction



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Language Basics

Terraform
Resource Syntax

Terraform
Resource Behavior

Terraform
State

Terraform
Resources

Resource
Meta-Argument
count

Resource
Meta-Argument
depends_on

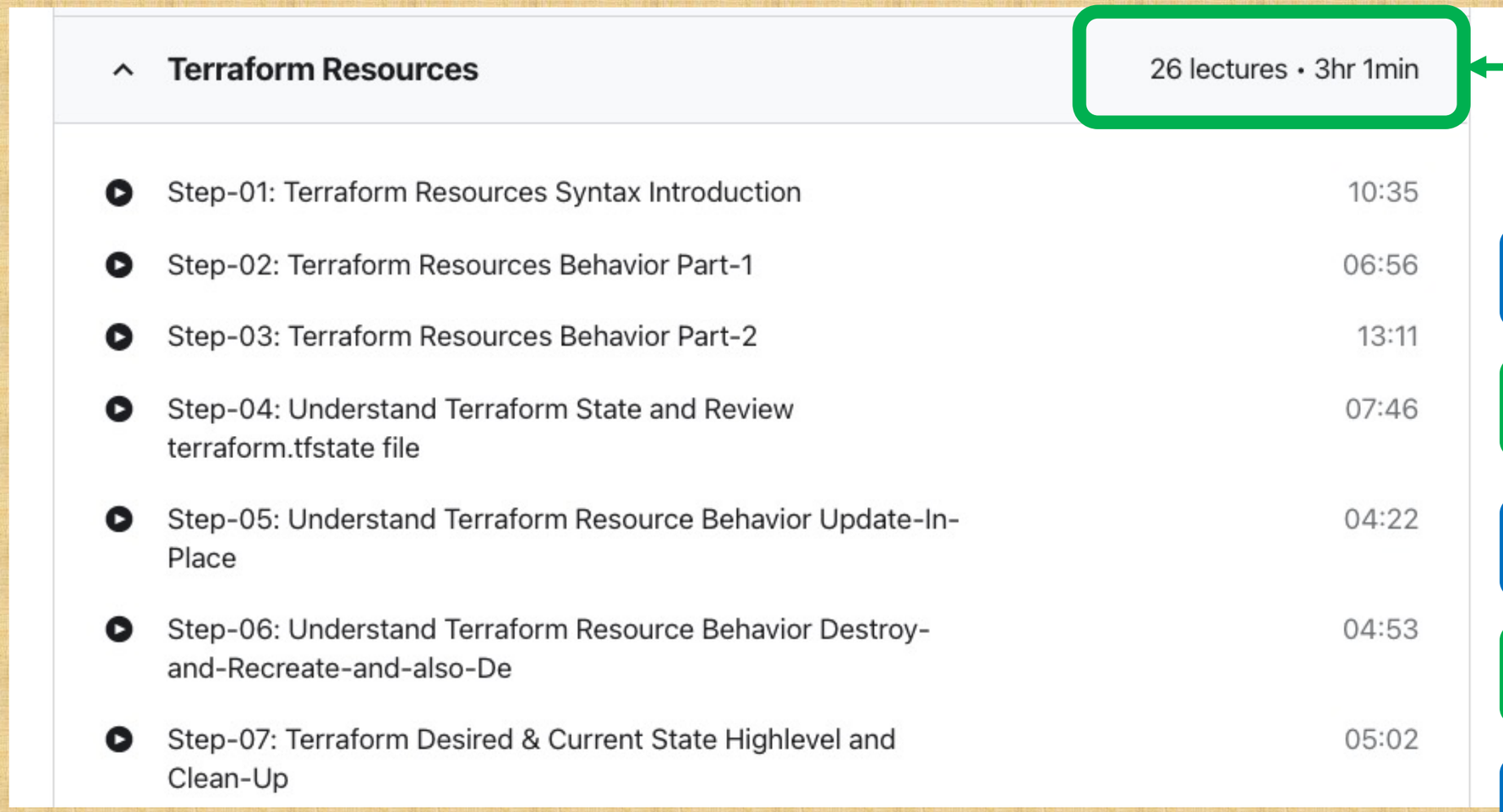
Resource
Meta-Argument
for_each

Resource
Meta-Argument
lifecycle

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Resources - Duration



WHY?

- Language Syntax
- Resource Syntax
- Resource Behavior
- State Fundamentals
- Resource Meta-Arguments

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>"   {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

# AWS Example
resource "aws_instance" "ec2demo" {
  ami           = "ami-04d29b6f966df1537"
  instance_type = "t2.micro"
}
```

Block Labels

Block Type

Top Level & Block inside Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners, resource specific blocks like tags

Arguments

Based on Block Type block labels will be 1 or 2
Example:
Resource – 2 labels
Variables – 1 label

Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Syntax

Resource Type: It determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports.

Resource Local Name: It 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

Meta-Arguments: Can be used with any resource to change the behavior of resources

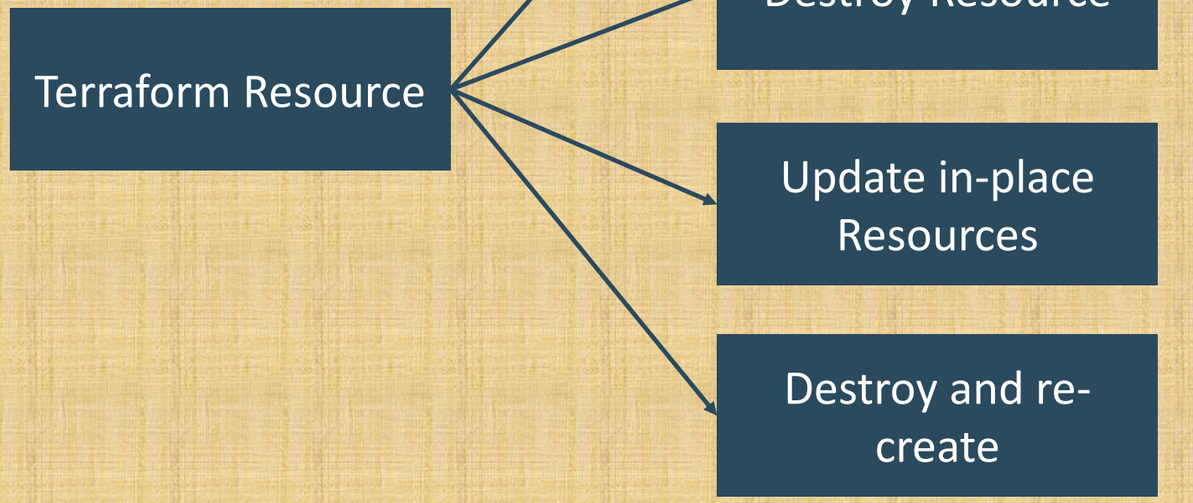
Resource Arguments: Will be specific to resource type. Argument Values can make use of Expressions or other Terraform Dynamic Language Features

```
# Provider-2 for us-west-1
provider "aws" {
  region = "us-west-1"
  profile = "default"
  alias = "aws-west-1"
}

# Resource Block to Create VPC
resource "aws_vpc" "vpc_us-west-1" {
  provider = aws.aws-west-1
  cidr_block = "10.2.0.0/16"
  tags = {
    "Name" = "vpc-1"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Behavior



Create Resource	Create resources that exist in the configuration but are not associated with a real infrastructure object in the state.
Destroy Resource	Destroy resources that exist in the state but no longer exist in the configuration.
Update in-place Resources	Update in-place resources whose arguments have changed.
Destroy and re-create	Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

Terraform State

Author: Nho Luong

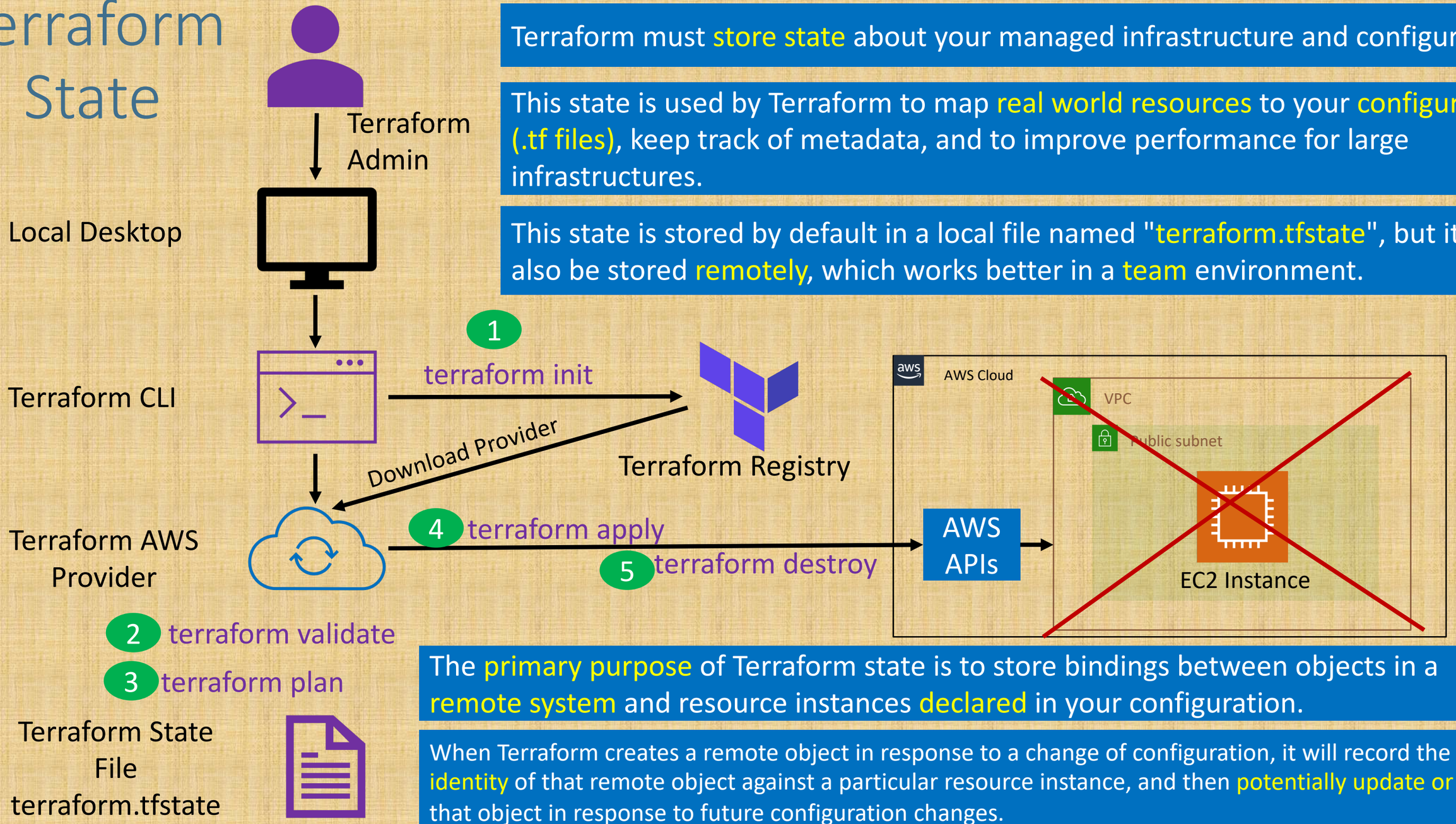
Skill: DevOps Engineer Lead

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 (.tf files), keep track of metadata, and to improve performance for large infrastructures.

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



The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.

When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

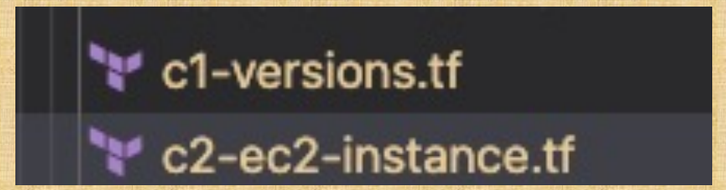
Author: Nho Luong

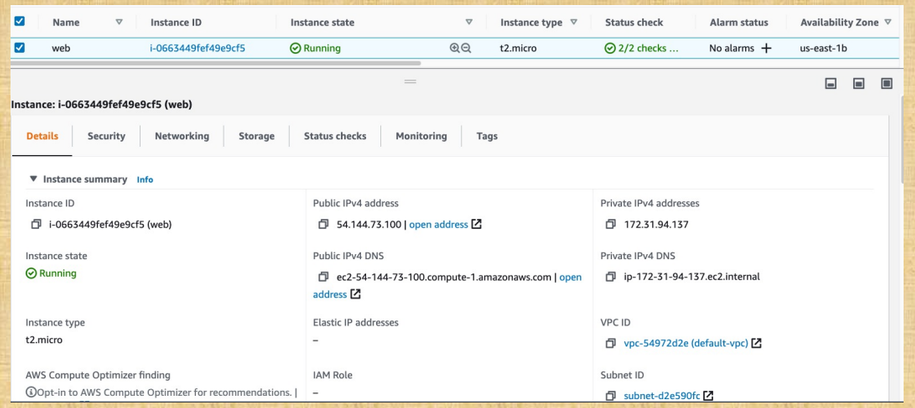
Skill: DevOps Engineer Lead

Desired & Current Terraform States

Terraform Configuration Files

Real World Resource – EC2 Instance





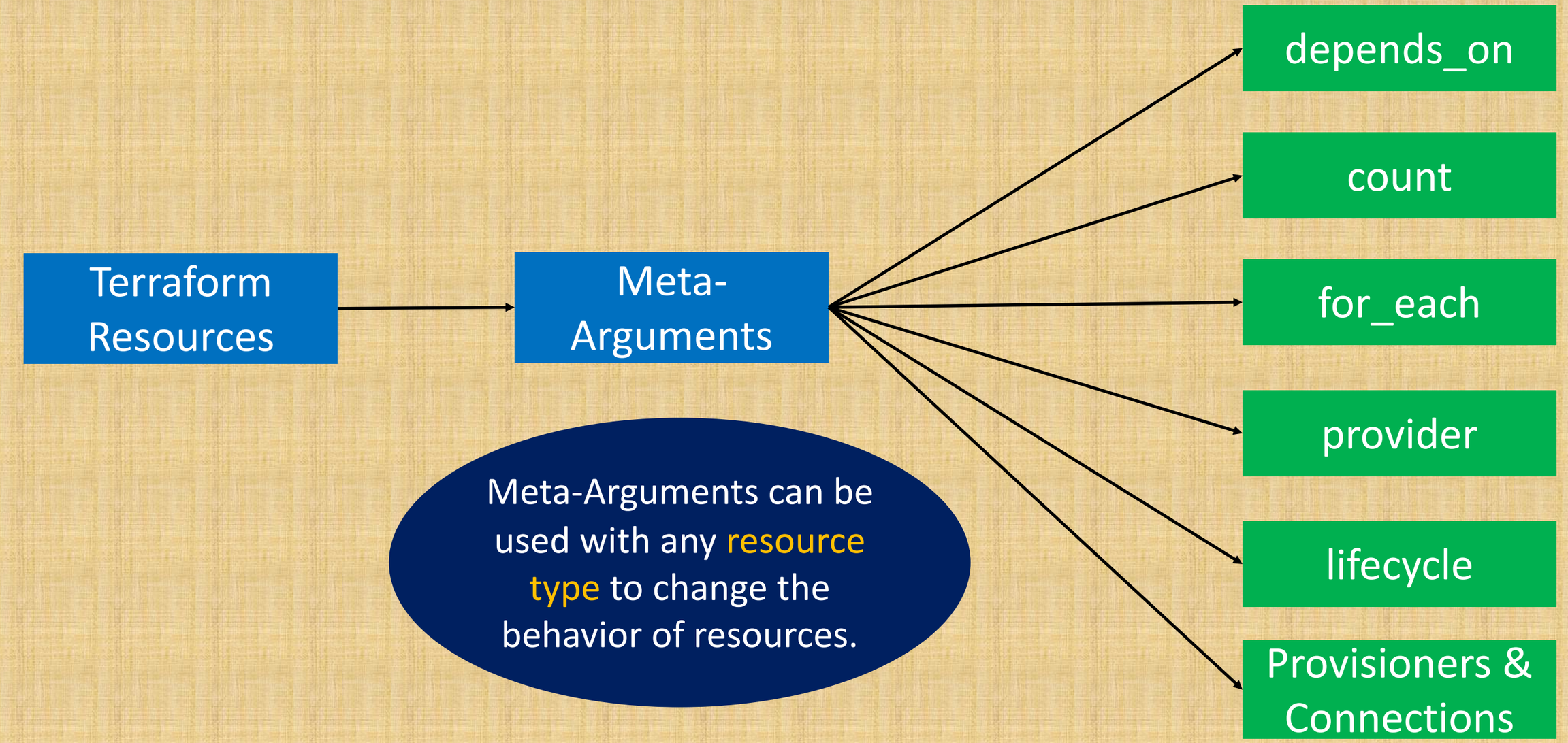
Desired State

Current State

Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Meta-Arguments



Author: Nho Luong

Skill: DevOps Engineer Lead

Use case: What are we going implement?

Resource-1: Create VPC

Resource-2: Create Subnets

Resource-3: Internet Gateway

Resource-4: Create Route Table

Resource-5: Create Route in Route Table for Internet Access

Resource-6: Associate the Route Table with the Subnet

Resource-7: Create Security Group

Resource-8: Create EC2 Instance with Sample App

EIP may require IGW to exist prior to association.
Use depends_on to set an explicit dependency on the IGW.

Resource-9: Create Elastic IP
depends_on

```
# Resource-9: Create Elastic IP
resource "aws_eip" "my-eip" {
  instance = aws_instance.my-ec2-vm.id
  vpc = true
  depends_on = [ aws_internet_gateway.vpc-dev-igw ]
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Usecase-1: for_each Maps

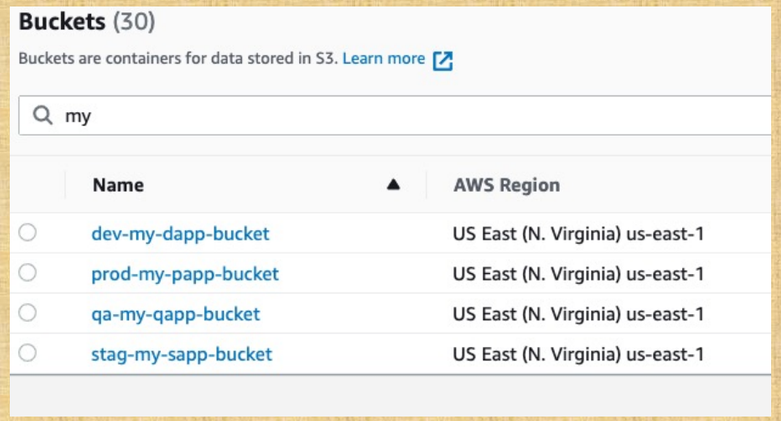
Resource-1: Use Meta-Argument for_each with Maps to create multiple S3 buckets using single Resource

Define for_each with Map with Key Value pairs

Use each.key and each.value for the S3 Bucket name

Use each.key and each.value for S3 Bucket tags

```
# Create S3 Bucket per environment with for_each and maps
resource "aws_s3_bucket" "mys3bucket" {

  for_each = {
    dev   = "my-dapp-bucket"
    qa    = "my-qapp-bucket"
    stag  = "my-sapp-bucket"
    prod  = "my-papp-bucket"
  }

  bucket = "${each.key}-${each.value}"
  acl    = "private"

  tags = {
    eachvalue   = each.value
    Environment = each.key
    bucketname  = "${each.key}-${each.value}"
  }
}
```



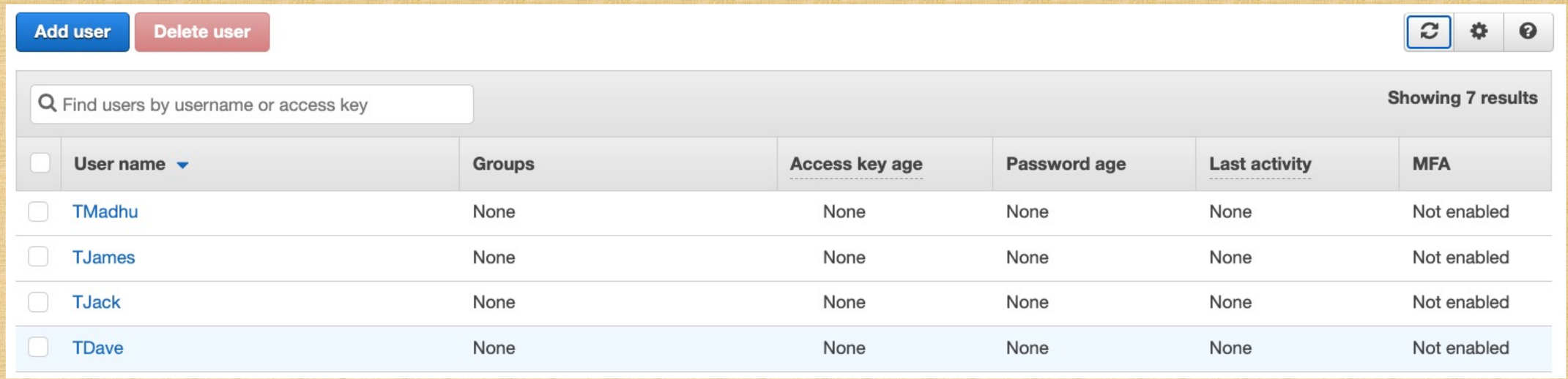
Author: Nho Luong

Skill: DevOps Engineer Lead

Usecase-2: for_each Set of Strings (toset)

Resource-1: Use Meta-Argument for_each with Set of Strings to create multiple IAM Users using single Resource

```
# Create 4 IAM Users
resource "aws_iam_user" "myuser" {
  for_each = toset( ["TJack", "TJames", "TMadhu", "TDave"] )
  name     = each.key
}
```



User name	Groups	Access key age	Password age	Last activity	MFA
TMadhu	None	None	None	None	Not enabled
TJames	None	None	None	None	Not enabled
TJack	None	None	None	None	Not enabled
TDave	None	None	None	None	Not enabled

Author: Nho Luong

Skill: DevOps Engineer Lead

lifecycle is a nested block that can appear within a resource block

Resource Meta-Argument lifecycle

The lifecycle block and its contents are meta-arguments, available for all resource blocks regardless of type.

create_before_destroy

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892" # A
  instance_type = "t2.micro"
  availability_zone = "us-east-1a"
  #availability_zone = "us-east-1b"
  tags = {
    "Name" = "web-1"
  }
  lifecycle {
    create_before_destroy = true
  }
}
```

prevent_destroy

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892"
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-2"
  }
  lifecycle {
    prevent_destroy = true # Def
  }
}
```

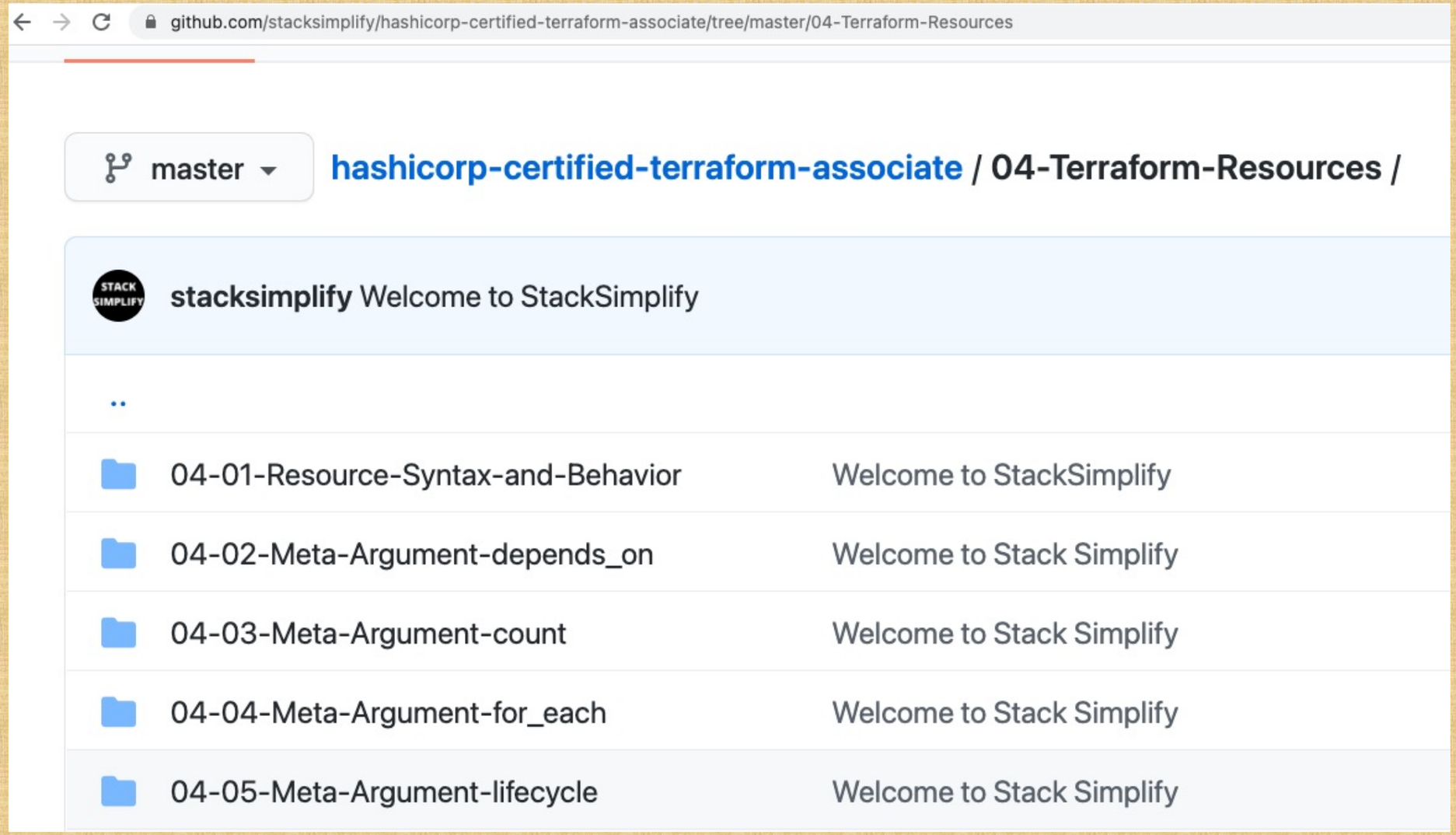
ignore_changes

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892"
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-3"
  }
  lifecycle {
    ignore_changes = [
      # Ignore changes to tags,
      # updates these based on
      tags,
    ]
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

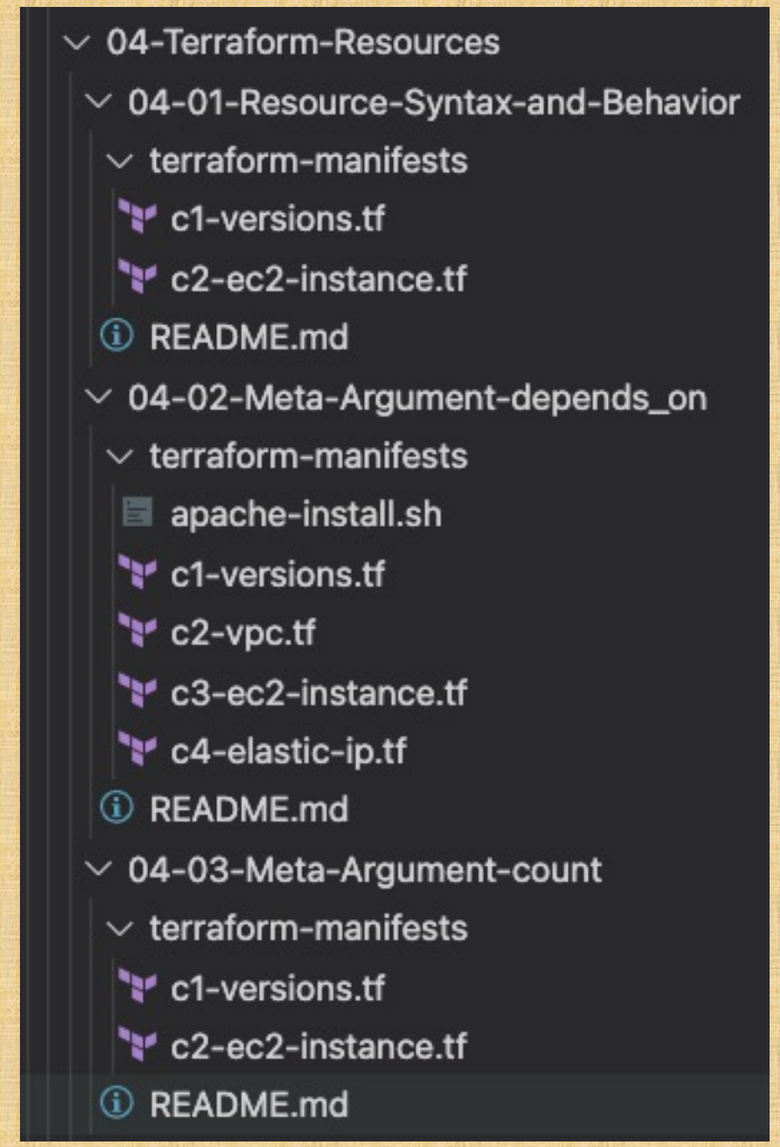
Github Step-by-Step Documentation

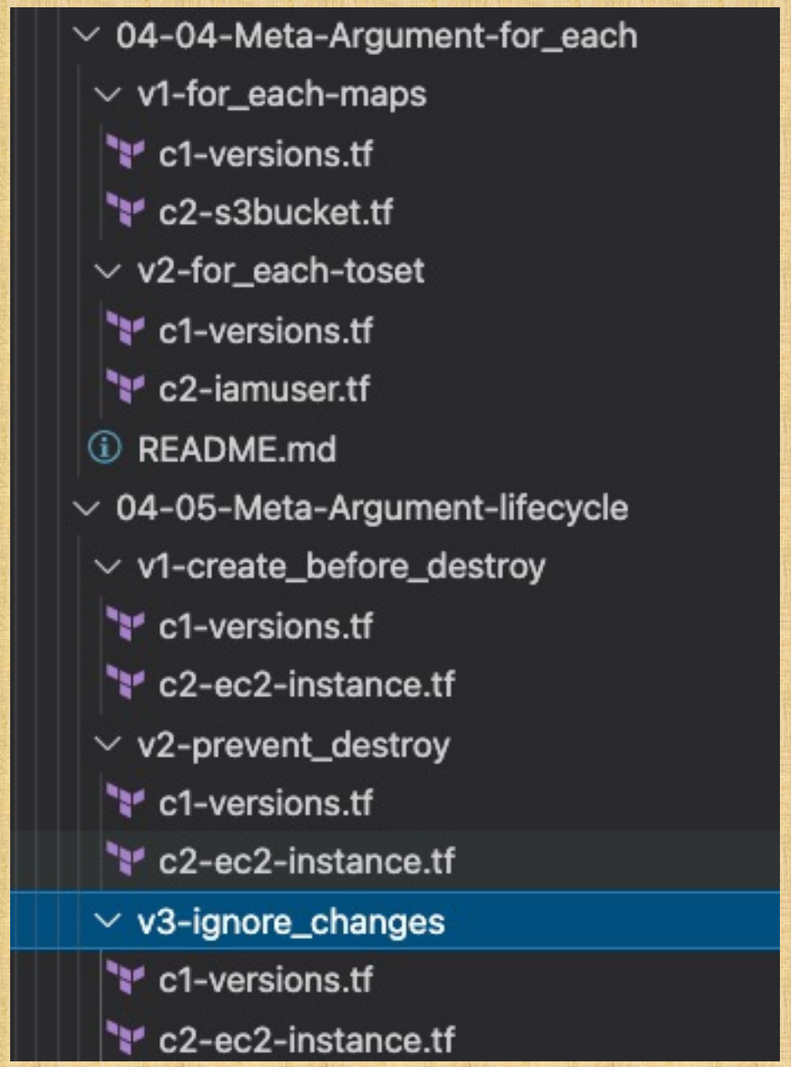


Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples For Each Concept



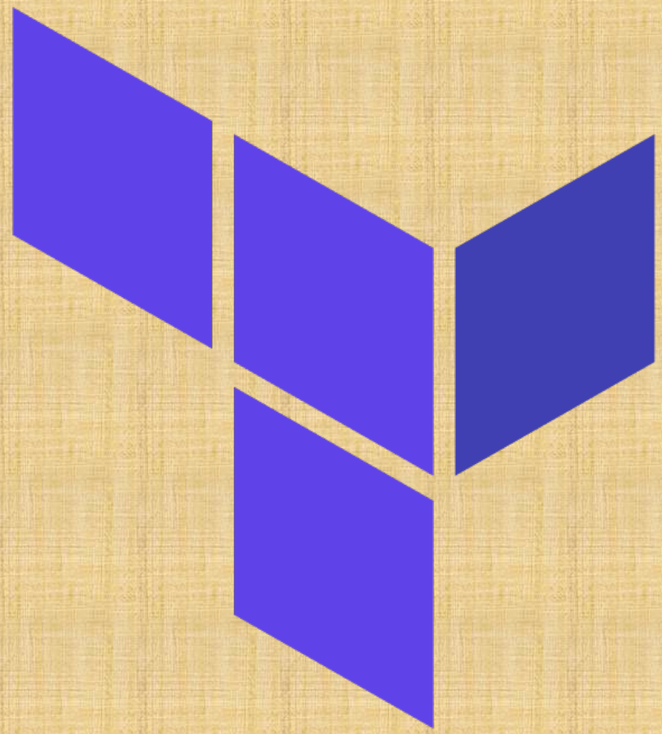


Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

Resource Syntax



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>"   {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

# AWS Example
resource "aws_instance" "ec2demo" {
  ami           = "ami-04d29b6f966df1537"
  instance_type = "t2.micro"
}
```

Block Labels

Block Type

Top Level & Block inside Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners, resource specific blocks like tags

Arguments

Based on Block Type block labels will be 1 or 2
Example:
Resource – 2 labels
Variables – 1 label

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Language Basics – Configuration Syntax

```
# Template
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>"   {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

# AWS Example
resource "aws_instance" "ec2demo" {
  ami           = "ami-04d29b6f966df1537"
  instance_type = "t2.micro"
}
```

Argument Name [or] Identifier

Argument Value [or] Expression

Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Syntax

Resource Type: It determines the kind of infrastructure object it manages and what arguments and other attributes the resource supports.

Resource Local Name: It 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

Meta-Arguments: Can be used with any resource to change the behavior of resources

Resource Arguments: Will be specific to resource type. Argument Values can make use of Expressions or other Terraform Dynamic Language Features

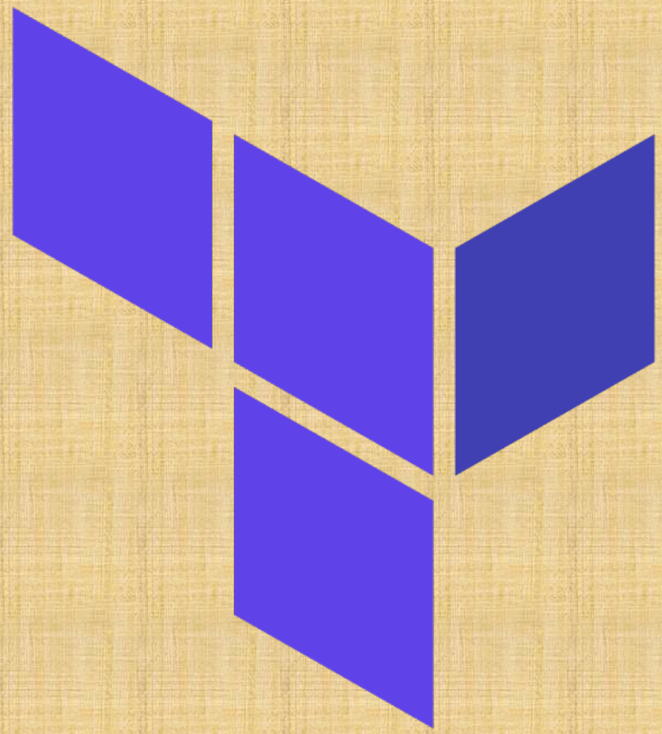
```
# Provider-2 for us-west-1
provider "aws" {
  region = "us-west-1"
  profile = "default"
  alias = "aws-west-1"
}

# Resource Block to Create VPC
resource "aws_vpc" "vpc_us-west-1" {
  provider = aws.aws-west-1
  cidr_block = "10.2.0.0/16"
  tags = {
    "Name" = "vpc-1"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

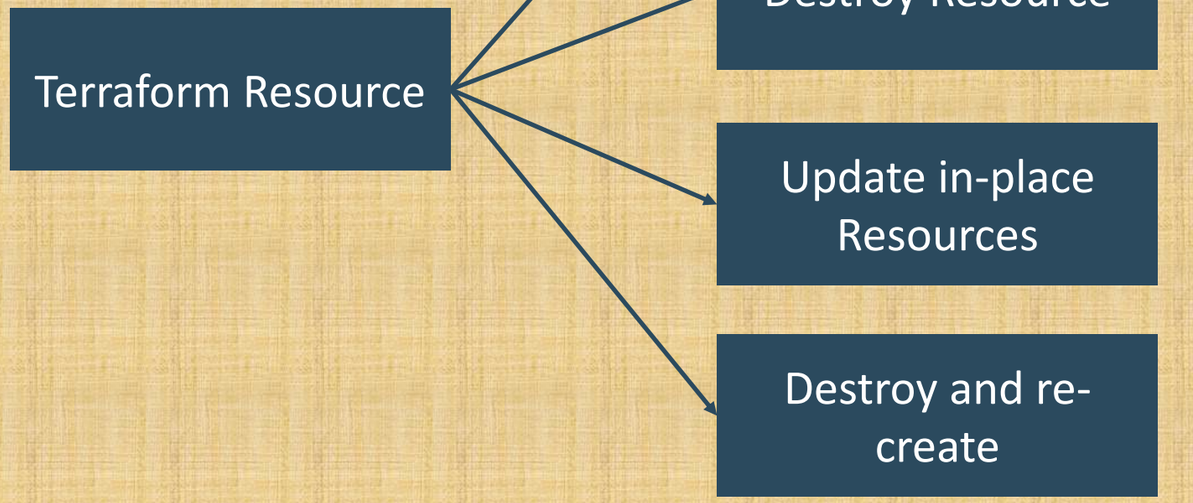
Resource Behavior



Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Behavior



Create Resource	Create resources that exist in the configuration but are not associated with a real infrastructure object in the state.
Destroy Resource	Destroy resources that exist in the state but no longer exist in the configuration.
Update in-place Resources	Update in-place resources whose arguments have changed.
Destroy and re-create	Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations.

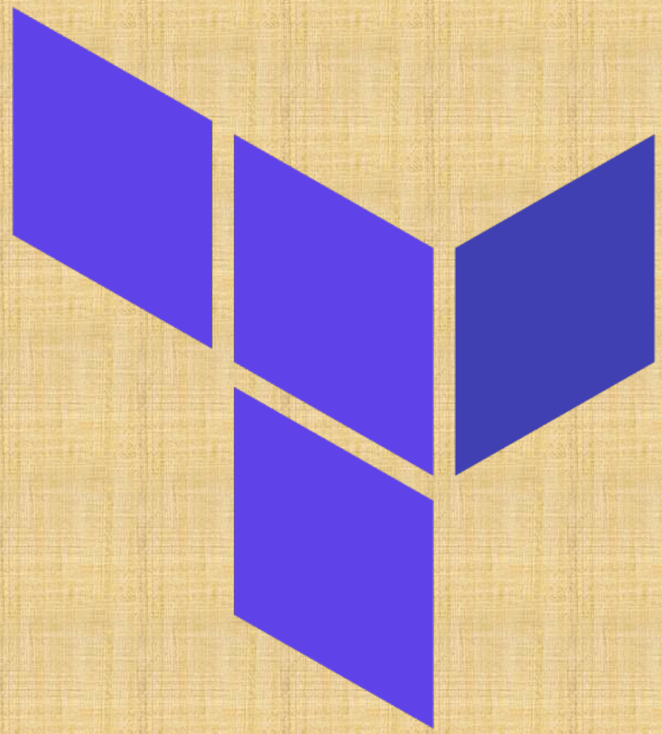
Terraform State

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

State



Author: Nho Luong

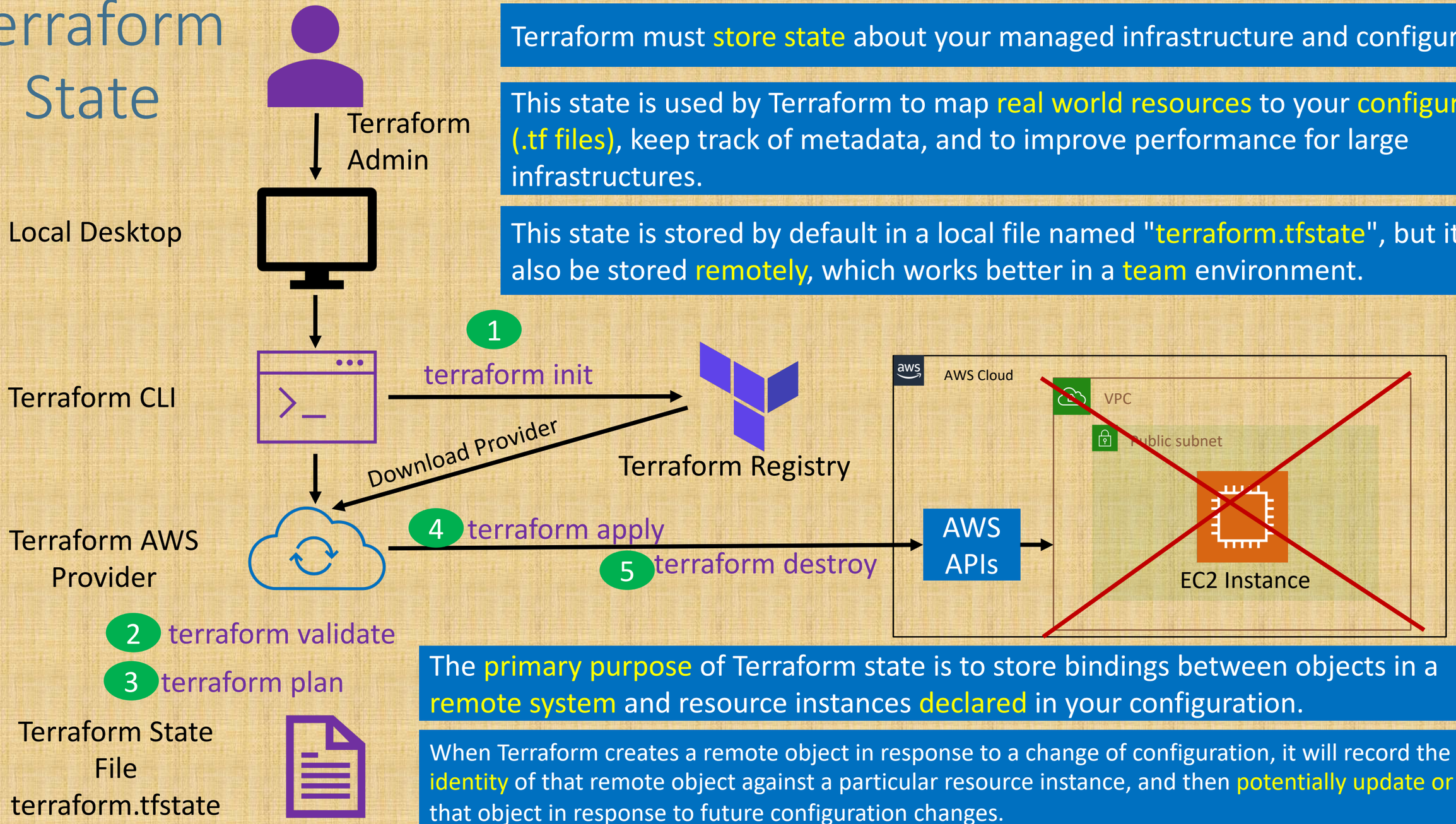
Skill: DevOps Engineer Lead

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 (.tf files), keep track of metadata, and to improve performance for large infrastructures.

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



The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.

When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

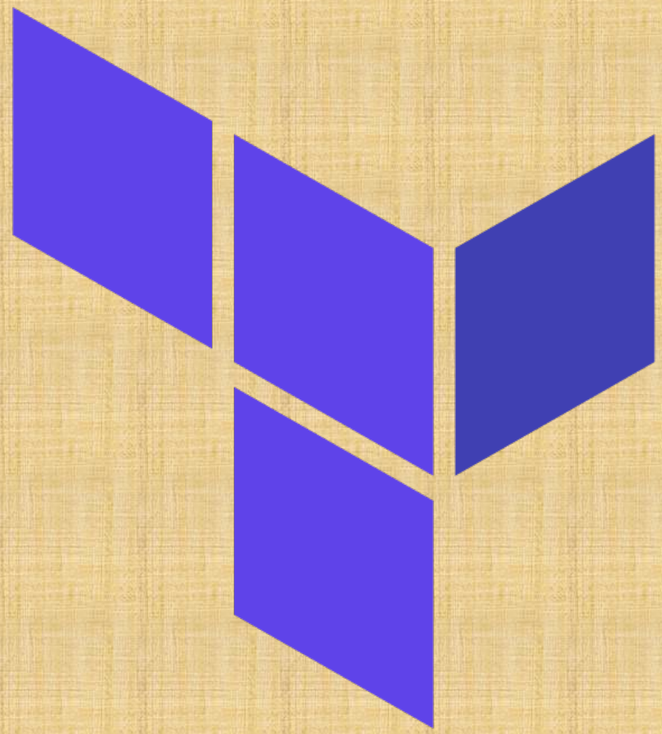
Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform

State

Desired & Current



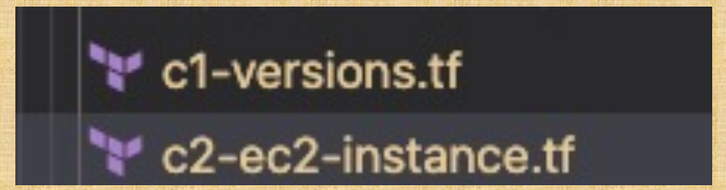
Author: Nho Luong

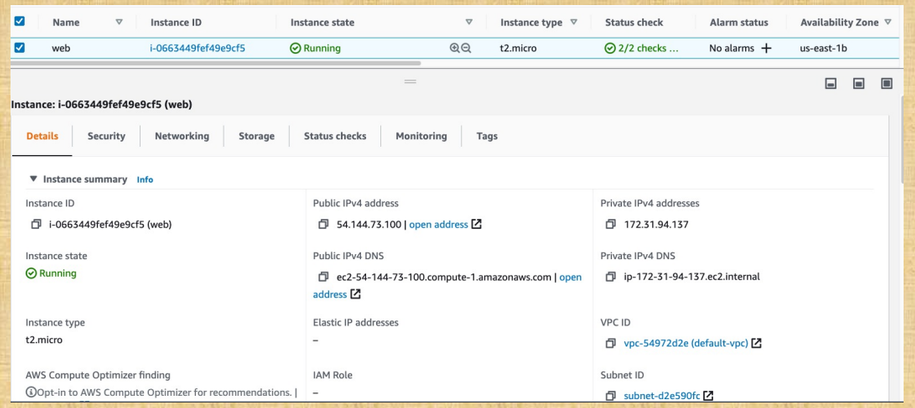
Skill: DevOps Engineer Lead

Desired & Current Terraform States

Terraform Configuration Files

Real World Resource – EC2 Instance





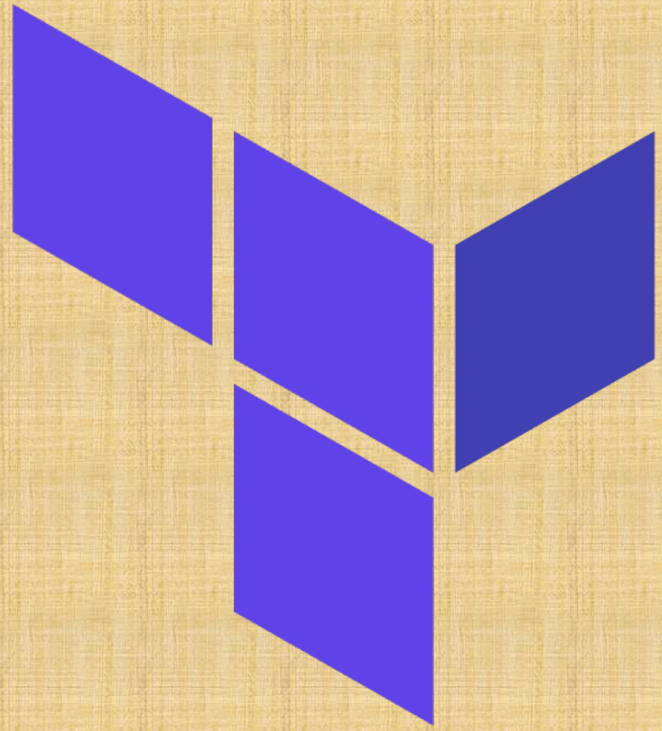
Desired State

Current State

Author: Nho Luong

Skill: DevOps Engineer Lead

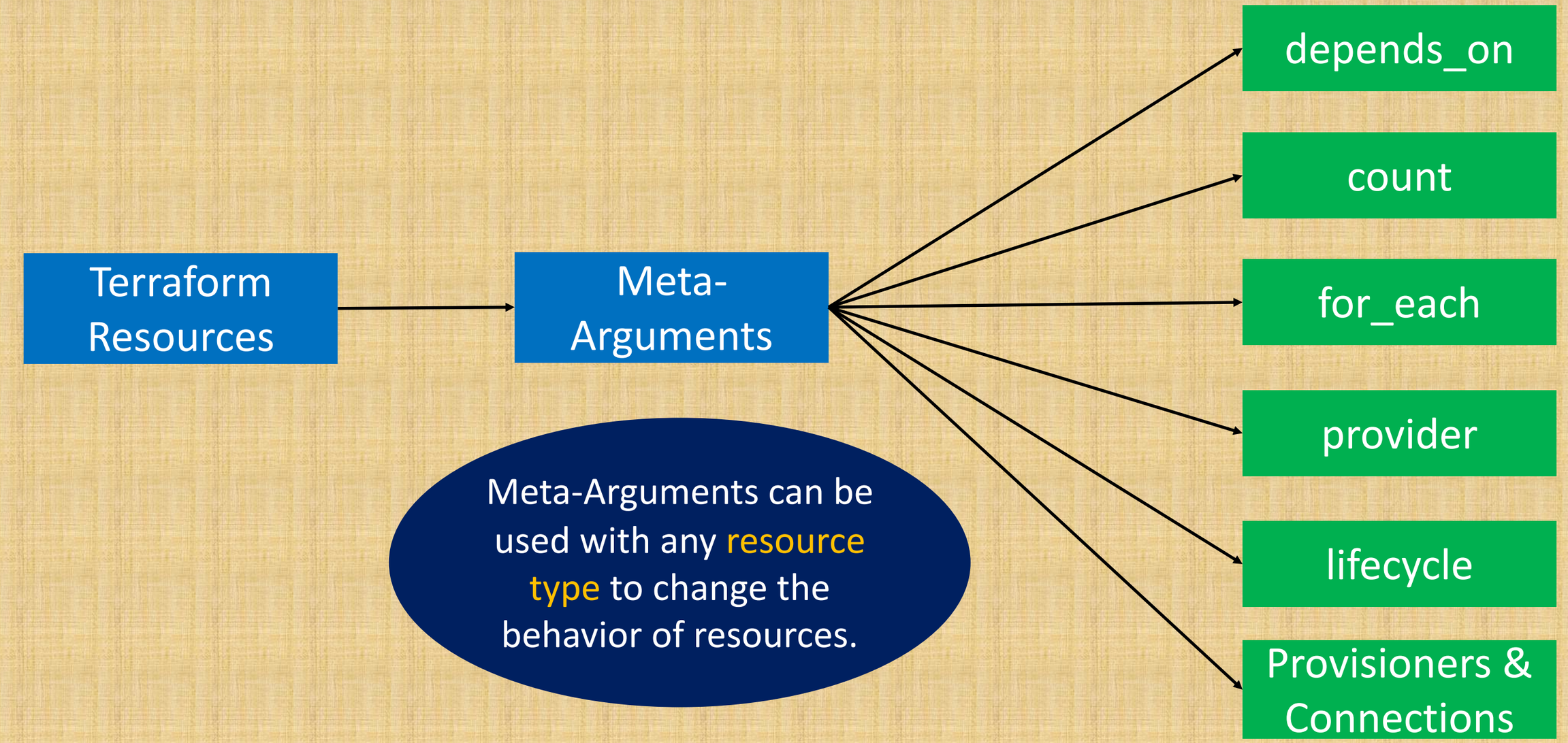
Terraform
Resource
Meta-Arguments



Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Meta-Arguments



Author: Nho Luong

Skill: DevOps Engineer Lead

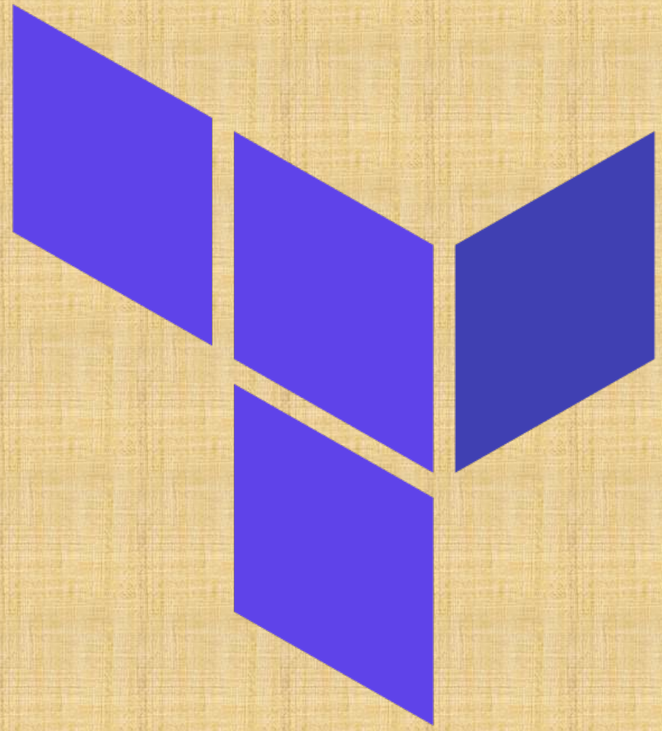
Resource Meta-Arguments

depends_on	To handle hidden resource or module dependencies that Terraform can't automatically infer.
count	For creating multiple resource instances according to a count
for_each	To create multiple instances according to a map, or set of strings
provider	For selecting a non-default provider configuration
lifecycle	Standard Resource behavior can be altered using special nested lifecycle block within a resource block body
Provisioners & Connections	For taking extra actions after resource creation (Example: install some app on server or do something on local desktop after resource is created at remote destination)

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Resource
Meta-Argument
depends_on



Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Meta-Arguments – depends_on

Use the depends_on meta-argument to handle hidden resource or module dependencies that Terraform can't automatically infer.

Explicitly specifying a dependency is only necessary when a resource or module relies on some other resource's behavior but *doesn't* access any of that resource's data in its arguments.

This argument is available in module blocks and in all resource blocks, regardless of resource type.

Resource Meta-Argument depends_on

The depends_on meta-argument, if present, must be a list of references to other resources or child modules in the same calling module.

Arbitrary expressions are not allowed in the depends_on argument value, because its value must be known before Terraform knows resource relationships and thus before it can safely evaluate expressions.

The depends_on argument should be used only as a last resort. Add comments for future reference about why we added this.

Author: Nho Luong

Skill: DevOps Engineer Lead

Use case: What are we going implement?

Resource-1: Create VPC

Resource-2: Create Subnets

Resource-3: Internet Gateway

Resource-4: Create Route Table

Resource-5: Create Route in Route Table for Internet Access

Resource-6: Associate the Route Table with the Subnet

Resource-7: Create Security Group

Resource-8: Create EC2 Instance with Sample App

EIP may require IGW to exist prior to association.
Use depends_on to set an explicit dependency on the IGW.

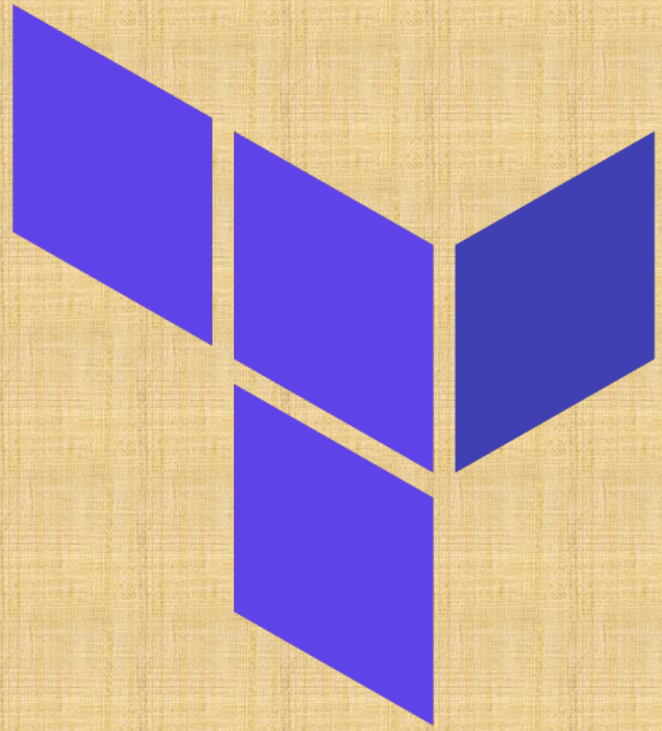
Resource-9: Create Elastic IP
depends_on

```
# Resource-9: Create Elastic IP
resource "aws_eip" "my-eip" {
  instance = aws_instance.my-ec2-vm.id
  vpc = true
  depends_on = [ aws_internet_gateway.vpc-dev-igw ]
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Resource
Meta-Argument
count



Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Meta-Arguments – count

If a resource or module block includes a count argument whose value is a whole number, Terraform will create that many instances.

count.index: The distinct index number (starting with 0) corresponding to this instance.

Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

Resource Meta-Argument count

When count is set, Terraform distinguishes between the block itself and the multiple resource or module instances associated with it. Instances are identified by an index number, starting with 0. aws_instance.myvm[0]

The count meta-argument accepts numeric expressions. The count value must be known *before* Terraform performs any remote resource actions.

Module support for count was added in Terraform 0.13, and previous versions can only use it with resources.

A given resource or module block cannot use both count and for_each

Author: Nho Luong

Skill: DevOps Engineer Lead

Use case: What are we going implement?

Resource-1: Use Meta-Argument Count to create multiple EC2 Instances using single Resource

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-047a51fa27710816e" # Amazon Linux
  instance_type = "t2.micro"
  count = 5
  tags = {
    #"Name" = "web"
    "Name" = "web-${count.index}"
  }
}
```

count

count.index

aws_instance.web[0]

aws_instance.web[1]

aws_instance.web[2]

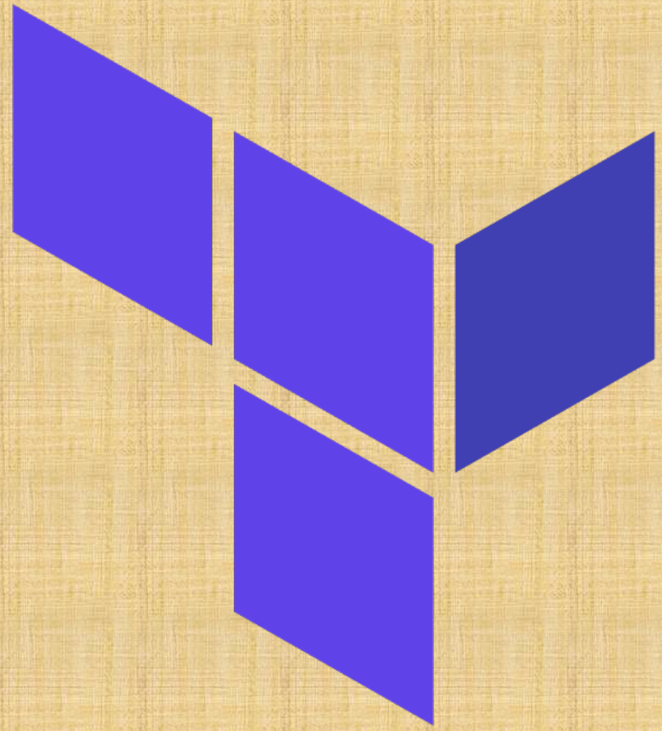
aws_instance.web[3]

aws_instance.web[4]

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Resource
Meta-Argument
for_each



Author: Nho Luong

Skill: DevOps Engineer Lead

Resource Meta-Arguments – for_each

If a resource or module block includes a for_each argument whose value is a map or a set of strings, Terraform will create one instance for each member of that map or set.

A given resource or module block cannot use both count and for_each

For set of Strings, each.key = each.value
for_each = toset(["Jack", "James"])
each.key = Jack
each.key = James

Each instance has a distinct infrastructure object associated with it, and each is separately created, updated, or destroyed when the configuration is applied.

Resource Meta-Argument for_each

For Maps, we use each.key & each.value
for_each = {
dev = "my-dapp-bucket"
}
each.key = dev
each.value = my-dapp-bucket

In blocks where for_each is set, an additional each object is available in expressions, so you can modify the configuration of each instance.
each.key — The map key (or set member) corresponding to this instance.
each.value — The map value corresponding to this instance. (If a set was provided, this is the same as each.key.)

Module support for for_each was added in Terraform 0.13, and previous versions can only use it with resources.

Author: Nho Luong

Skill: DevOps Engineer Lead

Usecase-1: for_each Maps

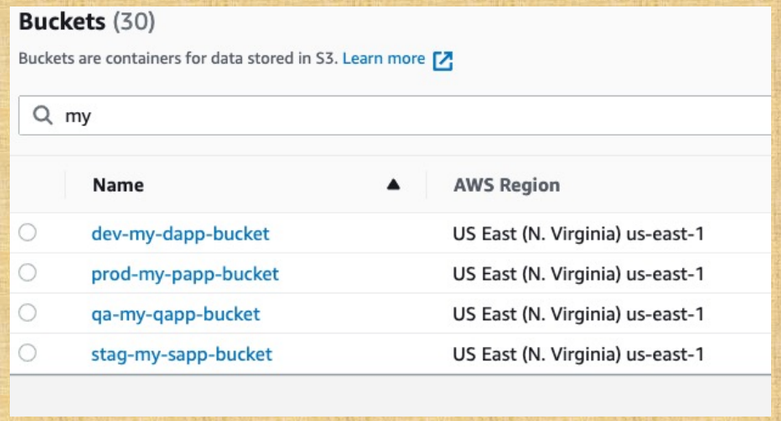
Resource-1: Use Meta-Argument for_each with Maps to create multiple S3 buckets using single Resource

Define for_each with Map with Key Value pairs

Use each.key and each.value for the S3 Bucket name

Use each.key and each.value for S3 Bucket tags

```
# Create S3 Bucket per environment with for_each and maps
resource "aws_s3_bucket" "mys3bucket" {

  for_each = {
    dev   = "my-dapp-bucket"
    qa    = "my-qapp-bucket"
    stag  = "my-sapp-bucket"
    prod  = "my-papp-bucket"
  }

  bucket = "${each.key}-${each.value}"
  acl    = "private"

  tags = {
    eachvalue   = each.value
    Environment = each.key
    bucketname  = "${each.key}-${each.value}"
  }
}
```



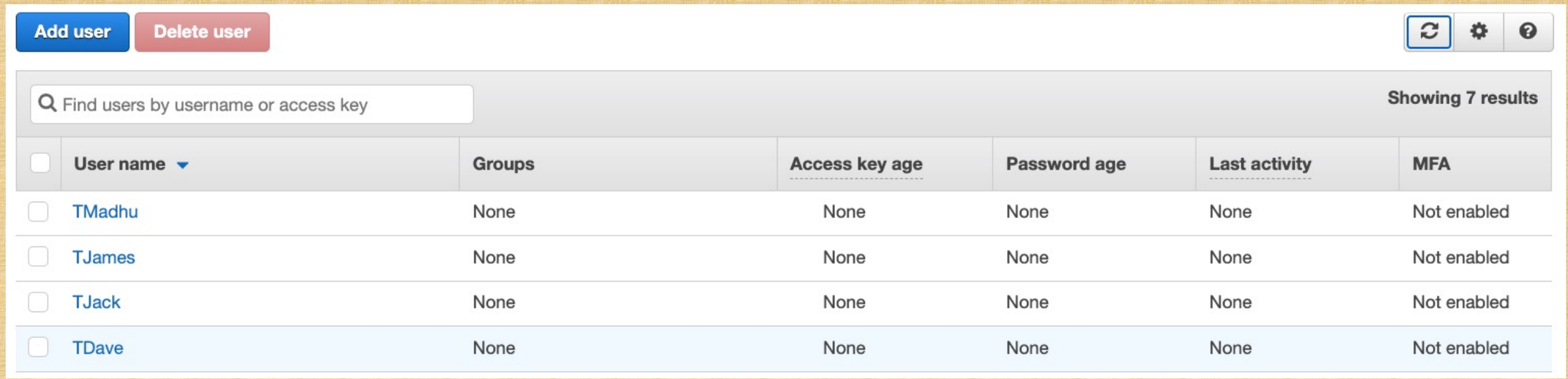
Author: Nho Luong

Skill: DevOps Engineer Lead

Usecase-2: for_each Set of Strings (toset)

Resource-1: Use Meta-Argument for_each with Set of Strings to create multiple IAM Users using single Resource

```
# Create 4 IAM Users
resource "aws_iam_user" "myuser" {
  for_each = toset( ["TJack", "TJames", "TMadhu", "TDave"] )
  name     = each.key
}
```

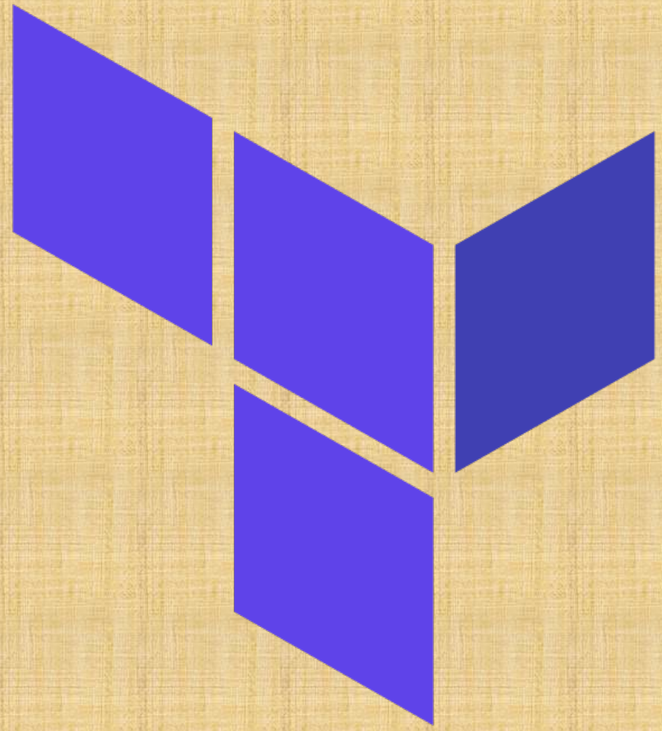


User name	Groups	Access key age	Password age	Last activity	MFA
TMadhu	None	None	None	None	Not enabled
TJames	None	None	None	None	Not enabled
TJack	None	None	None	None	Not enabled
TDave	None	None	None	None	Not enabled

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Resource
Meta-Argument
lifecycle



Author: Nho Luong

Skill: DevOps Engineer Lead

lifecycle is a nested block that can appear within a resource block

Resource Meta-Argument lifecycle

The lifecycle block and its contents are meta-arguments, available for all resource blocks regardless of type.

create_before_destroy

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892" # A
  instance_type = "t2.micro"
  availability_zone = "us-east-1a"
  #availability_zone = "us-east-1b"
  tags = {
    "Name" = "web-1"
  }
  lifecycle {
    create_before_destroy = true
  }
}
```

prevent_destroy

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892"
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-2"
  }
  lifecycle {
    prevent_destroy = true # Def
  }
}
```

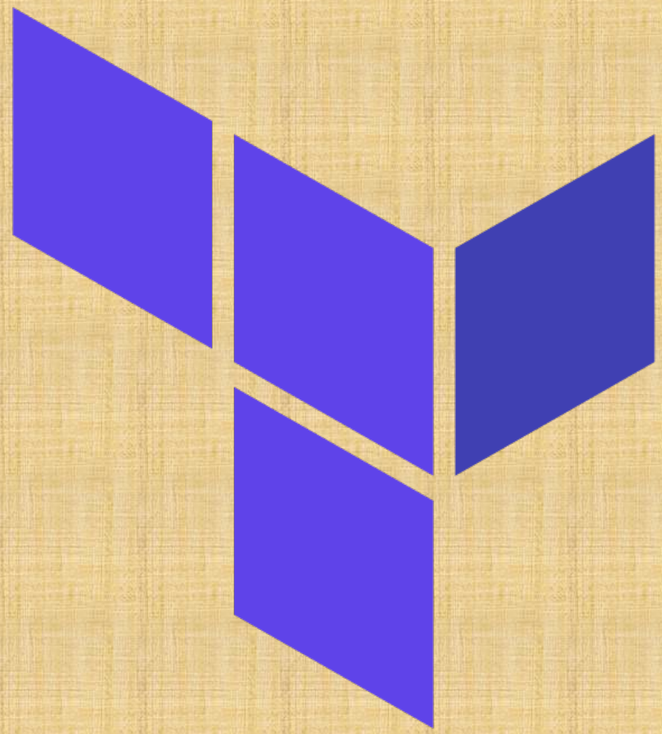
ignore_changes

```
# Create EC2 Instance
resource "aws_instance" "web" {
  ami = "ami-0915bcb5fa77e4892"
  instance_type = "t2.micro"
  tags = {
    "Name" = "web-3"
  }
  lifecycle {
    ignore_changes = [
      # Ignore changes to tags,
      # updates these based on
      tags,
    ]
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Variables
Introduction



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables

Terraform Input Variables

Terraform Output Values

Terraform Local Values

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

Terraform Input Variables

1. Input Variables - Basics
2. Provide Input Variables when prompted during terraform plan or apply
3. Override default variable values using CLI argument -var
4. Override default variable values using Environment Variables (TF_var_aa)
5. Provide Input Variables using terraform.tfvars files
6. Provide Input Variables using <any-name>.tfvars file with CLI argument -var-file
7. Provide Input Variables using auto.tfvars files
8. Implement complex type constructors like List & Map in Input Variables
9. Implement Custom Validation Rules in Variables
10. Protect Sensitive Input Variables

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables – Output Values

Output values are like the return values of a Terraform module and have several uses

Terraform Variables Outputs

1

A root module can use outputs to print certain values in the CLI output after running terraform apply.

2

A child module can use outputs to expose a subset of its resource attributes to a parent module.

Advanced

3

When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state d ata source.

Author: Nho Luong

Skill: DevOps Engineer Lead

DRY Principle

Don’t Repeat Yourself

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables – Local Values

A local value assigns a name to an expression, so you can use that name multiple times within a module without repeating it.

Local values are like a function's temporary local variables.

Once a local value is declared, you can reference it in expressions as local.<NAME>.

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration

If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used

The ability to easily change the value in a central place is the key advantage of local values.

In short, Use local values only in moderation

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

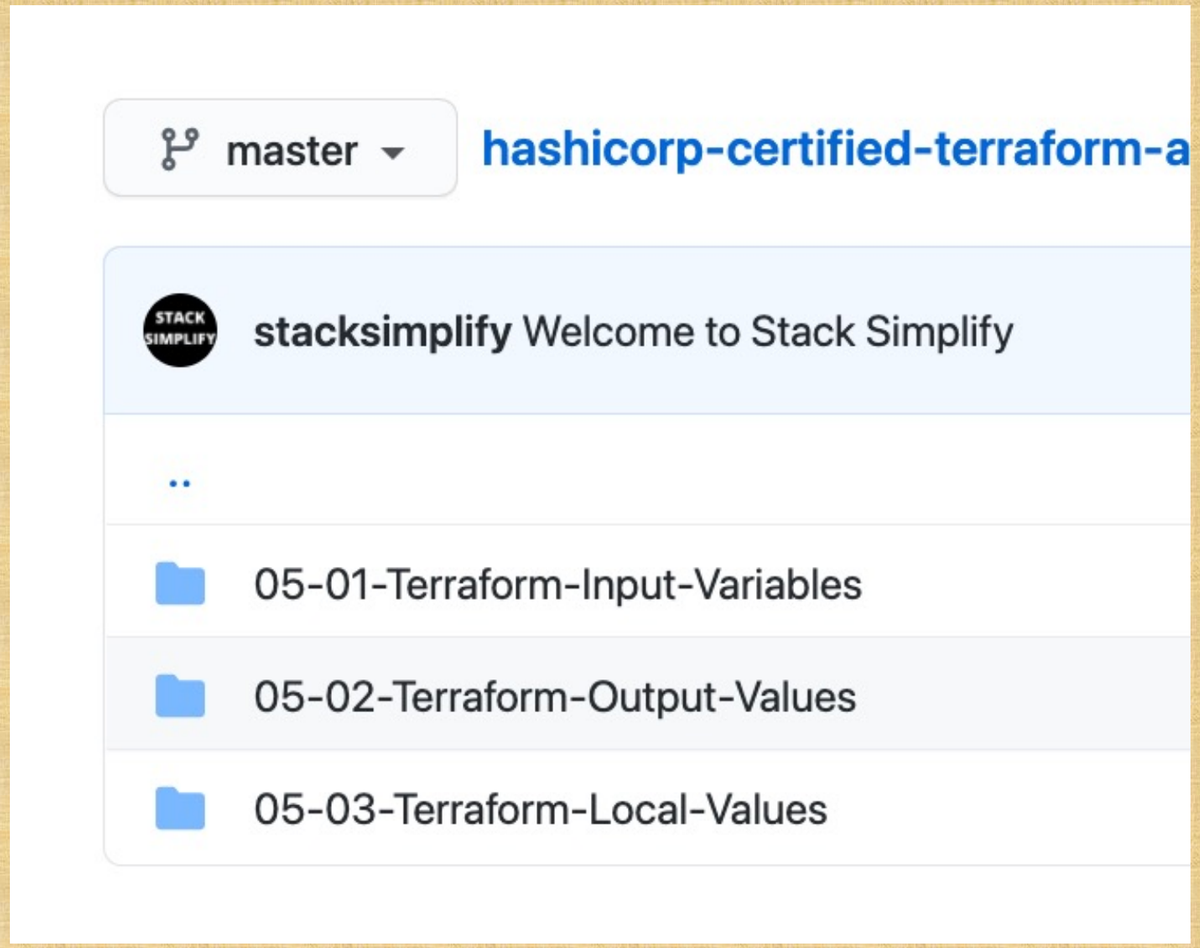
```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

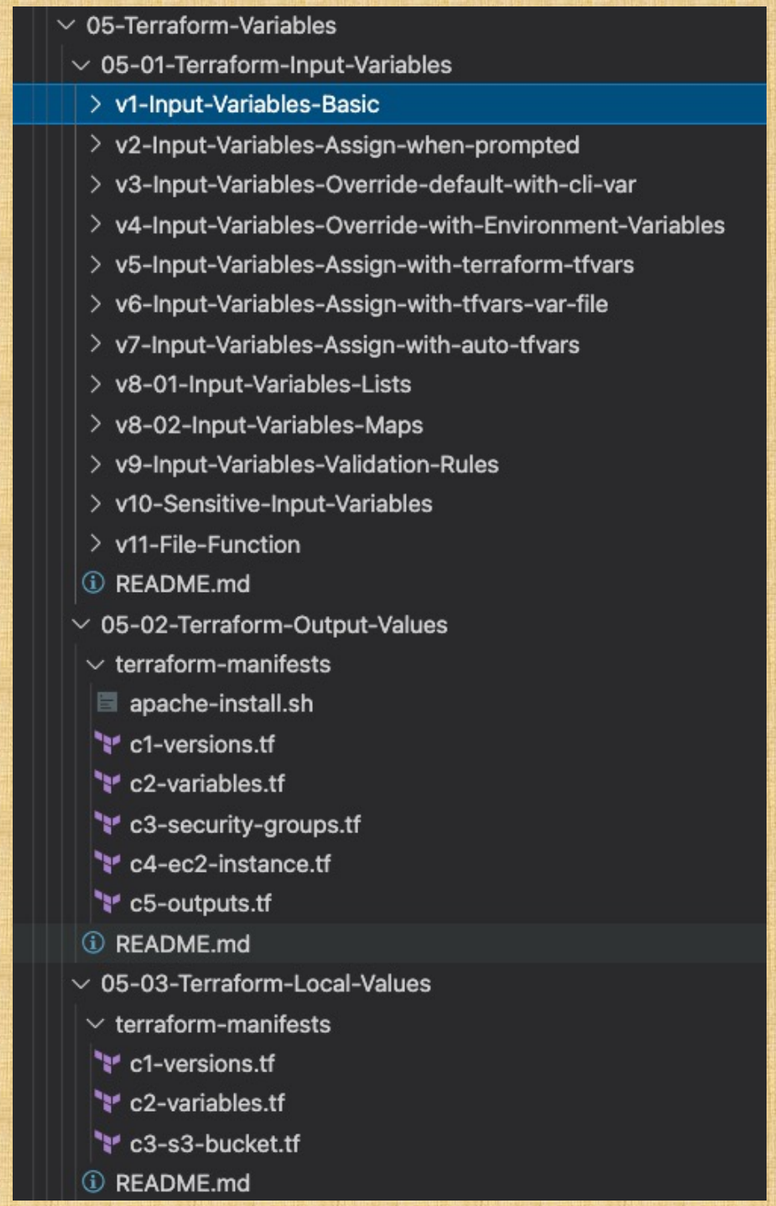
```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples with Step-by-Step Documentation on Github





Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables - Duration

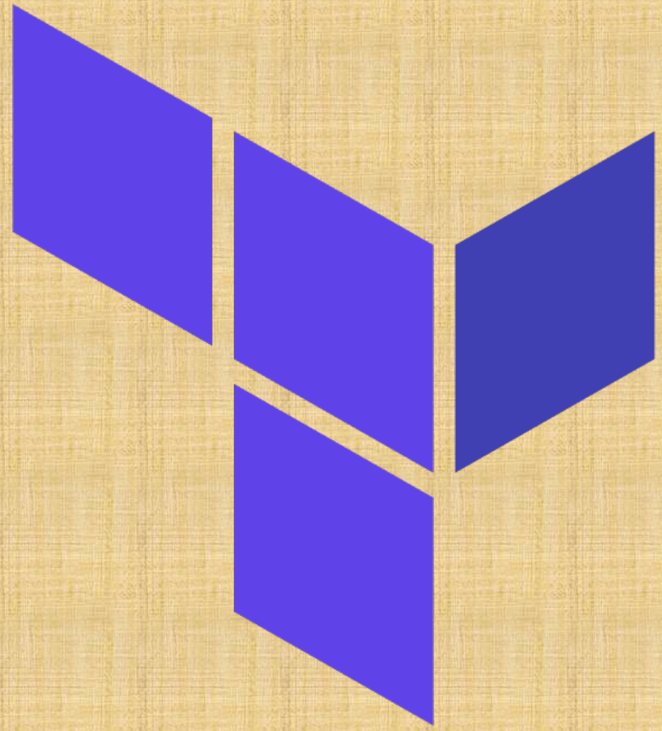
Terraform Variables — 21 lectures • 2hr 28min

Step-01: Terraform Input Variables Introduction	04:47
Step-02: Terraform Input Variable Basics - Part-1	12:47
Step-03: Terraform Input Variable Basics - Part-2	11:33
Step-04: Input Variables - Assing When Prompted using CLI	04:29
Step-05: Input Variables - Override default value with -var argument	08:21
Step-06: Input Variables - Override default value with environment vairables	05:26
Step-07: Input Variables - Assign with terraform.tfvars	05:49
Step-08: Input Variables - Assign with -var-file argument	07:03
Step-09: Input Variables - Assign with .auto.tfvars files	04:19

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Variables
Input Variables



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

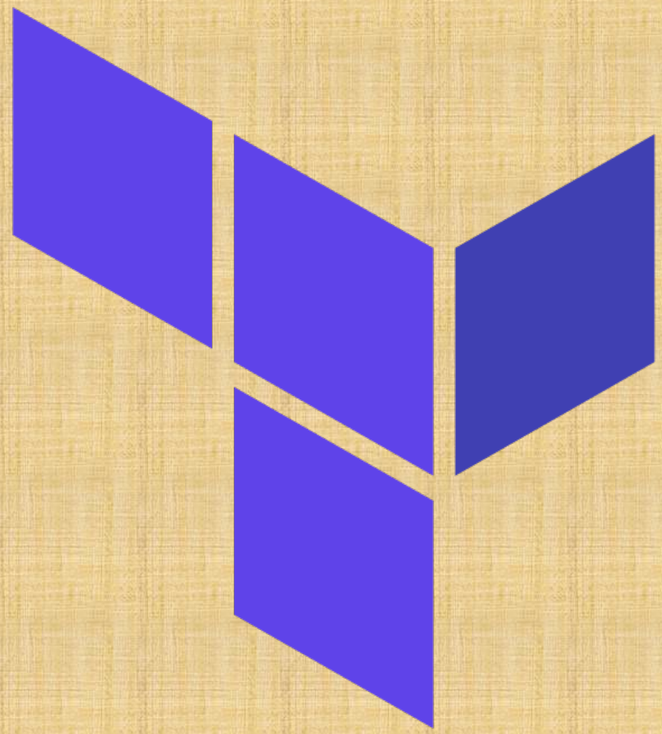
Terraform Input Variables

1. Input Variables - Basics
2. Provide Input Variables when prompted during terraform plan or apply
3. Override default variable values using CLI argument -var
4. Override default variable values using Environment Variables (TF_var_aa)
5. Provide Input Variables using terraform.tfvars files
6. Provide Input Variables using <any-name>.tfvars file with CLI argument -var-file
7. Provide Input Variables using auto.tfvars files
8. Implement complex type constructors like List & Map in Input Variables
9. Implement Custom Validation Rules in Variables
10. Protect Sensitive Input Variables

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Variables
Output Values



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables – Output Values

Output values are like the return values of a Terraform module and have several uses

Terraform Variables Outputs

1

A root module can use outputs to print certain values in the CLI output after running terraform apply.

2

A child module can use outputs to expose a subset of its resource attributes to a parent module.

Advanced

3

When using remote state, root module outputs can be accessed by other configurations via a terraform_remote_state d ata source.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables
Output Values

```
# Define Output Values
# Attribute Reference: EC2 Instance Public IP
output "ec2_instance_publicip" {
  description = "EC2 Instance Public IP"
  value = aws_instance.my-ec2-vm.public_ip
}

# Argument Reference: EC2 Instance Private IP
output "ec2_instance_privateip" {
  description = "EC2 Instance Private IP"
  value = aws_instance.my-ec2-vm.private_ip
}
# Argument Reference: Security Groups associated to EC2 Instance
output "ec2_security_groups" {
  description = "List Security Groups associated with EC2 Instance"
  value = aws_instance.my-ec2-vm.security_groups
}

# Attribute Reference - Create Public DNS URL with http:// appended
output "ec2_publicdns" {
  description = "Public DNS URL of an EC2 Instance"
  value = "http://${aws_instance.my-ec2-vm.public_dns}"
  #sensitive = true   #Uncomment it during step-04 execution
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

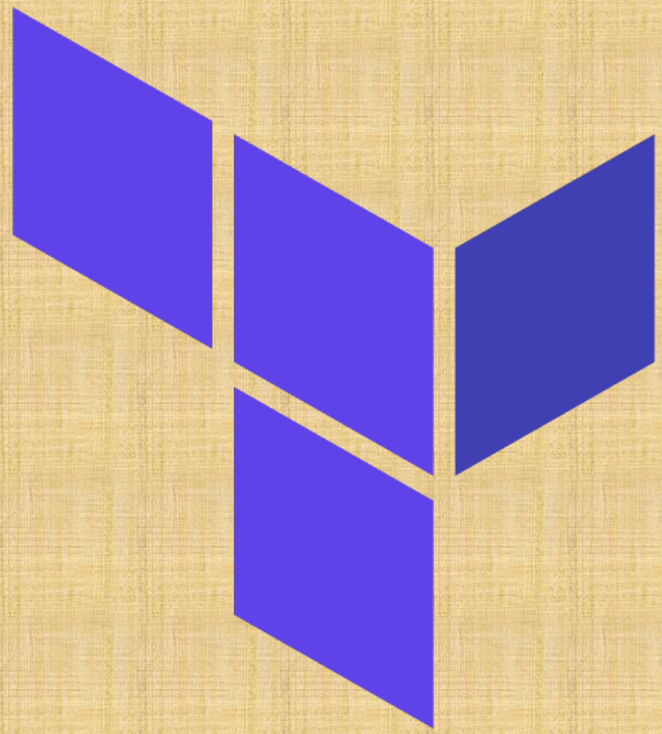
Terraform Variables - Output Values

```
aws_instance.my-ec2-vm: Creation complete after 24s [id=i-0406c6733

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

ec2_instance_privateip = "172.31.78.184"
ec2_instance_publicip = "3.235.244.111"
ec2_publicdns = "http://ec2-3-235-244-111.compute-1.amazonaws.com"
ec2_security_groups = toset([
  "vpc-ssh",
  "vpc-web",
])
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Variables
Local Values



Author: Nho Luong

Skill: DevOps Engineer Lead

DRY Principle

Don’t Repeat Yourself

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Variables – Local Values

A local value assigns a name to an expression, so you can use that name multiple times within a module without repeating it.

Local values are like a function's temporary local variables.

Once a local value is declared, you can reference it in expressions as local.<NAME>.

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration

If overused they can also make a configuration hard to read by future maintainers by hiding the actual values used

The ability to easily change the value in a central place is the key advantage of local values.

In short, Use local values only in moderation

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

```
locals {
  # Common tags to be assigned to all resources
  common_tags = {
    Service = local.service_name
    Owner   = local.owner
  }
}
```

```
resource "aws_instance" "example" {
  # ...

  tags = local.common_tags
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

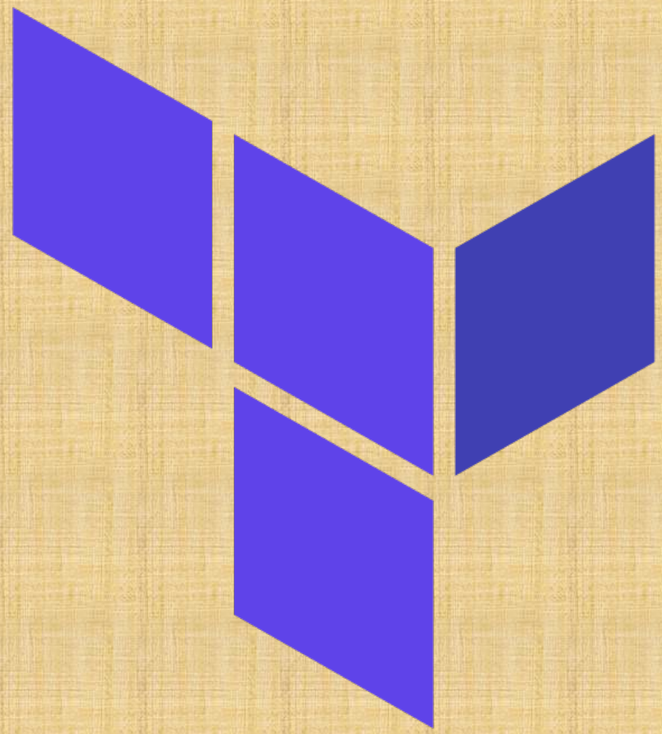
Terraform Variables – Local Values

```
# Create S3 Bucket - with Input Variables & Local Values
locals {
  bucket-name = "${var.app_name}-${var.environment_name}-bucket" # Complex expression
}

resource "aws_s3_bucket" "mys3bucket" {
  bucket = local.bucket-name # Simplifed to use in many places
  acl = "private"
  tags = {
    Name = local.bucket-name # Simplifed to use in many places
    Environment = var.environment_name
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Datasources



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Datasources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.

Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

A data source is accessed via a special kind of resource known as a *data resource*, declared using a data block

Each data resource is associated with a single data source, which determines the kind of object (or objects) it reads and what query constraint arguments are available

Data resources have the same dependency resolution behavior as defined for managed resources. Setting the depends_on meta-argument within data blocks defers reading of the data source until after all changes to the dependencies have been applied.

```
# Get latest AMI ID for Amazon Linux2 OS
data "aws_ami" "amzlinux" {
  most_recent      = true
  owners           = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-hvm-*"]
  }
  filter {
    name   = "root-device-type"
    values = ["ebs"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  filter {
    name   = "architecture"
    values = ["x86_64"]
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Datasources

We can refer the data resource in a resource as depicted

```
# Create EC2 Instance - Amazon Linux
resource "aws_instance" "my-ec2-vm" {
  ami                    = data.aws_ami.amzlinux.id
  instance_type = var.ec2_instance_type
  key_name       = "terraform-key"
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group
  tags = {
    "Name" = "amz-linux-vm"
  }
}
```

Meta-Arguments for Datasources

Data resources support the provider meta-argument as defined for managed resources, with the same syntax and behavior.

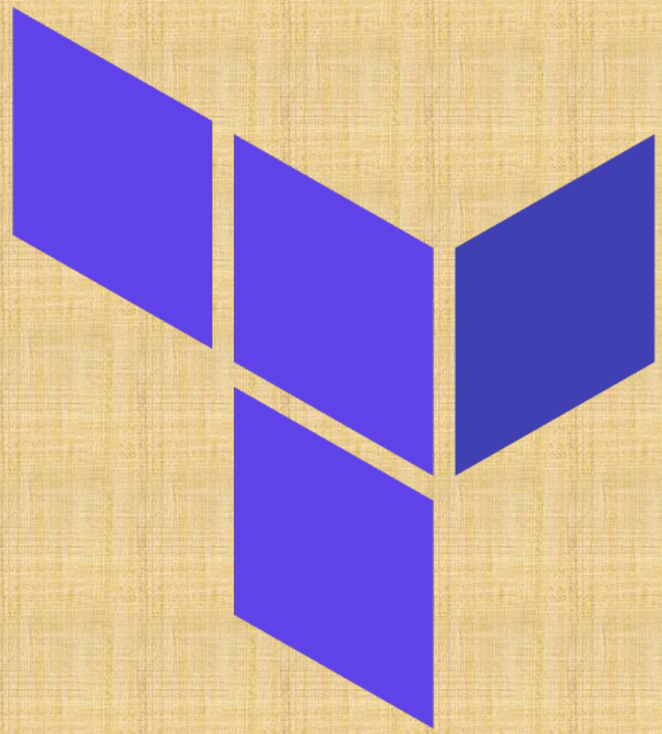
Data resources do not currently have any customization settings available for their lifecycle, but the lifecycle nested block is reserved in case any are added in future versions.

Data resources support count and for_each meta-arguments as defined for managed resources, with the same syntax and behavior.
Each instance will separately read from its data source with its own variant of the constraint arguments, producing an indexed result.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
State
Introduction



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform State

Terraform
Remote
State
Storage

Terraform
Commands
from
State
Perspective

Author: Nho Luong

Skill: DevOps Engineer Lead

What is Terraform Backend ?

Backends are responsible for storing state and providing an API for state locking.

Terraform
State Storage

AWS S3 Bucket

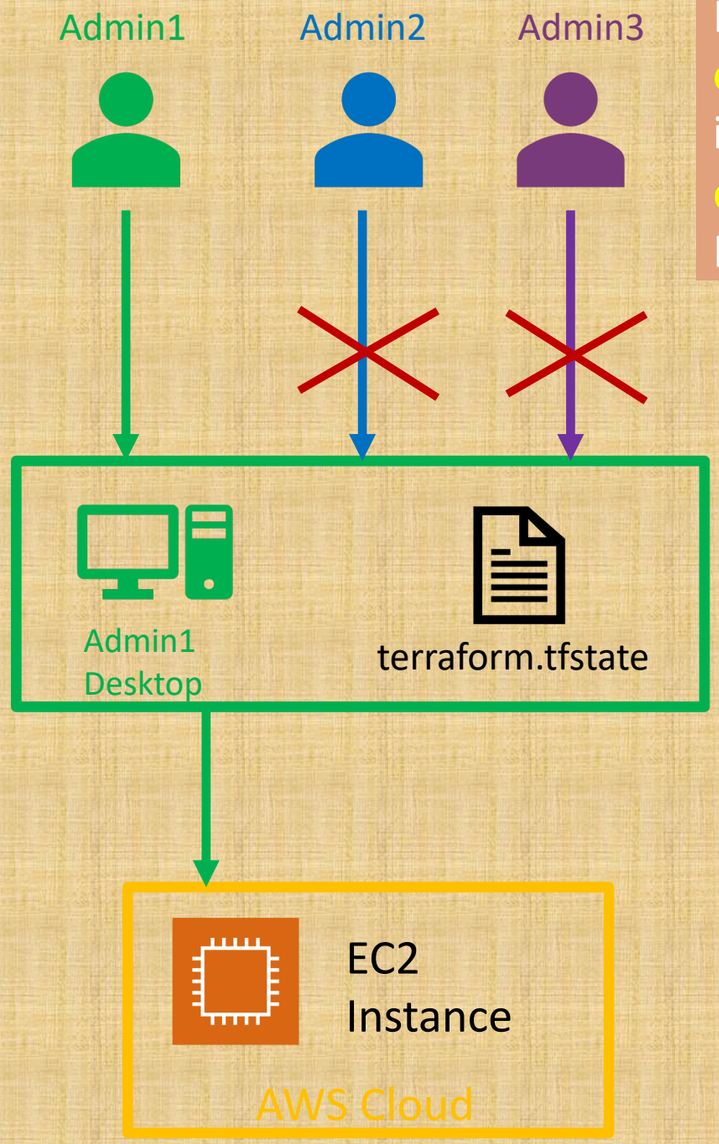
Terraform
State Locking

AWS DynamoDB

Author: Nho Luong

Skill: DevOps Engineer Lead

Local State File



Multiple Team members cannot update the infrastructure as they don't have access to State File

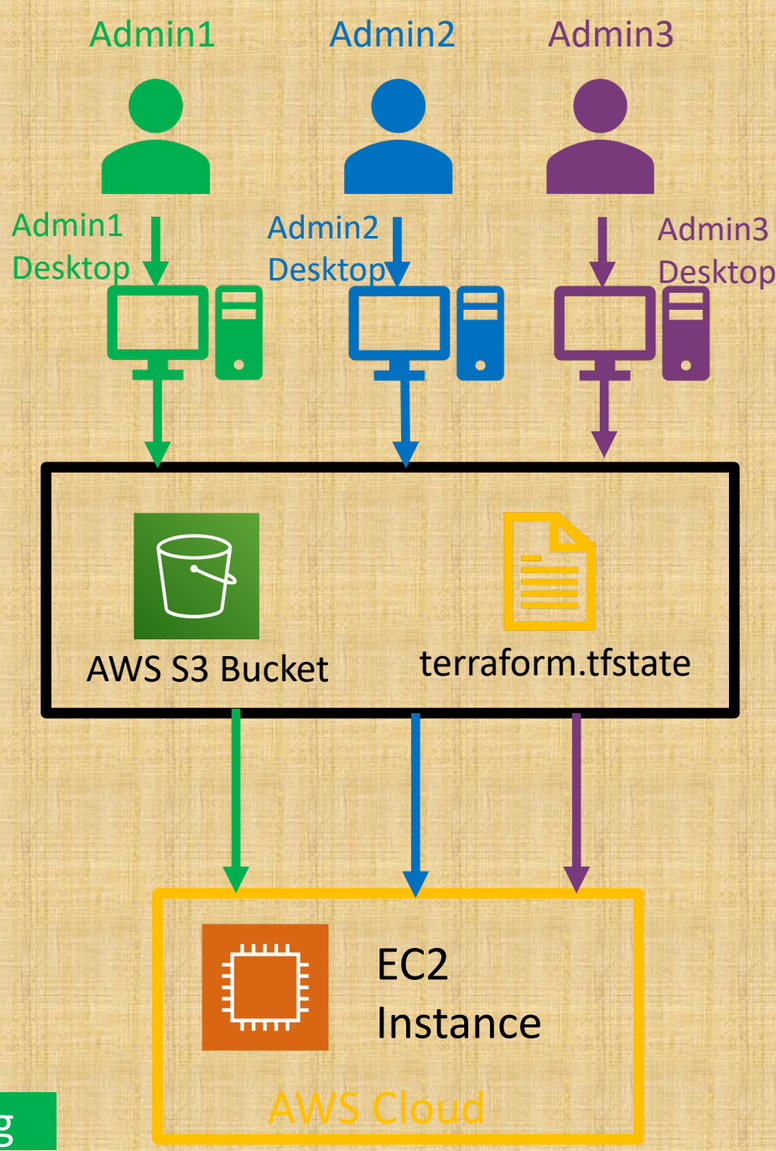
This means we need store the state file in a shared location.

Remote State File

Using Terraform Backend concept we can use AWS S3 as the shared storage for State Files

If two team members are running Terraform at the same time, you may run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.

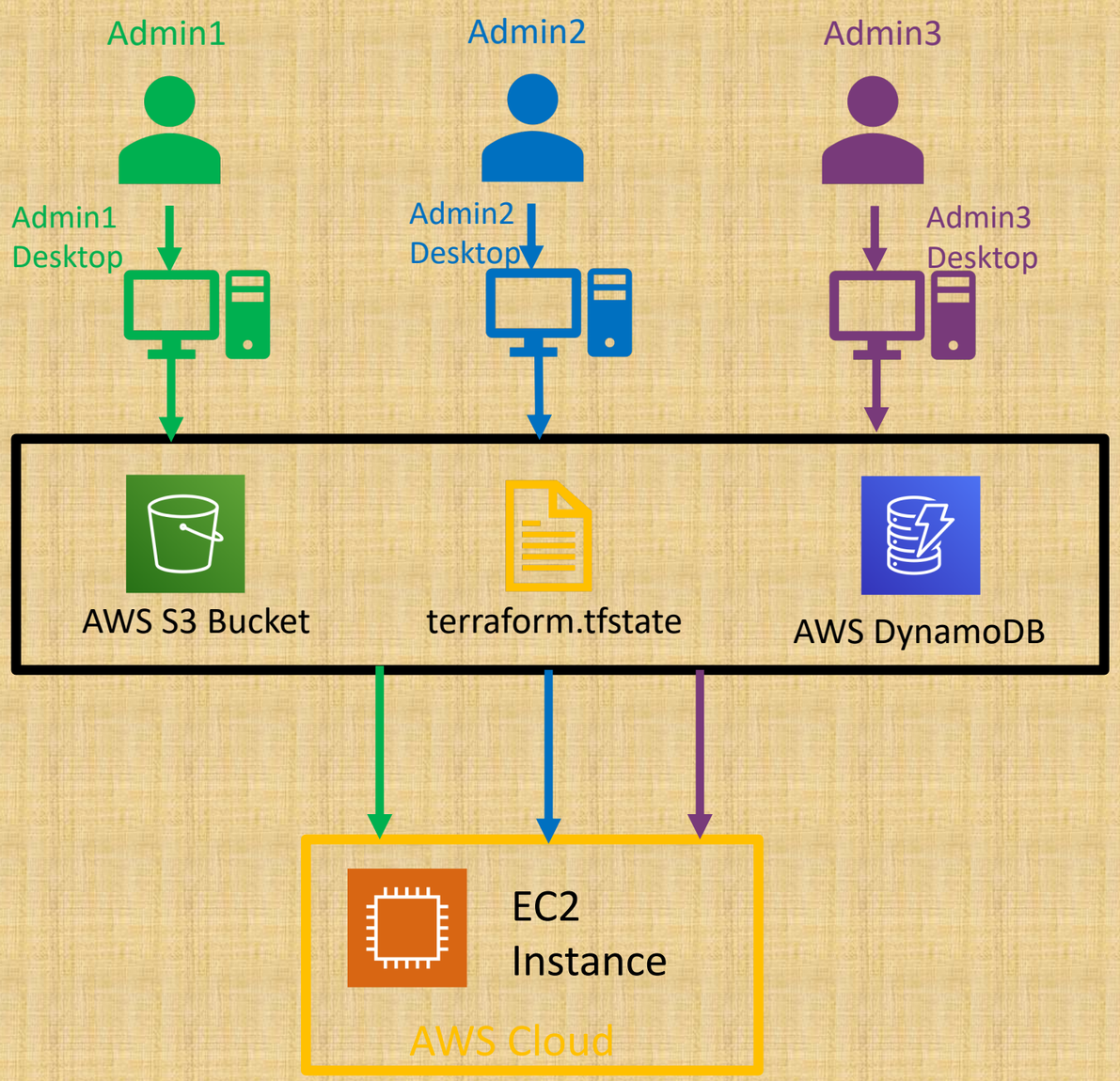
Implement State Locking



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Remote State File with State Locking



Not all backends support State Locking. AWS S3 supports State Locking

State locking happens automatically on all operations that could write state.

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.

If acquiring the lock is taking longer than expected, Terraform will output a status message.

If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Remote State File with State Locking

Terraform State Storage to Remote Backend

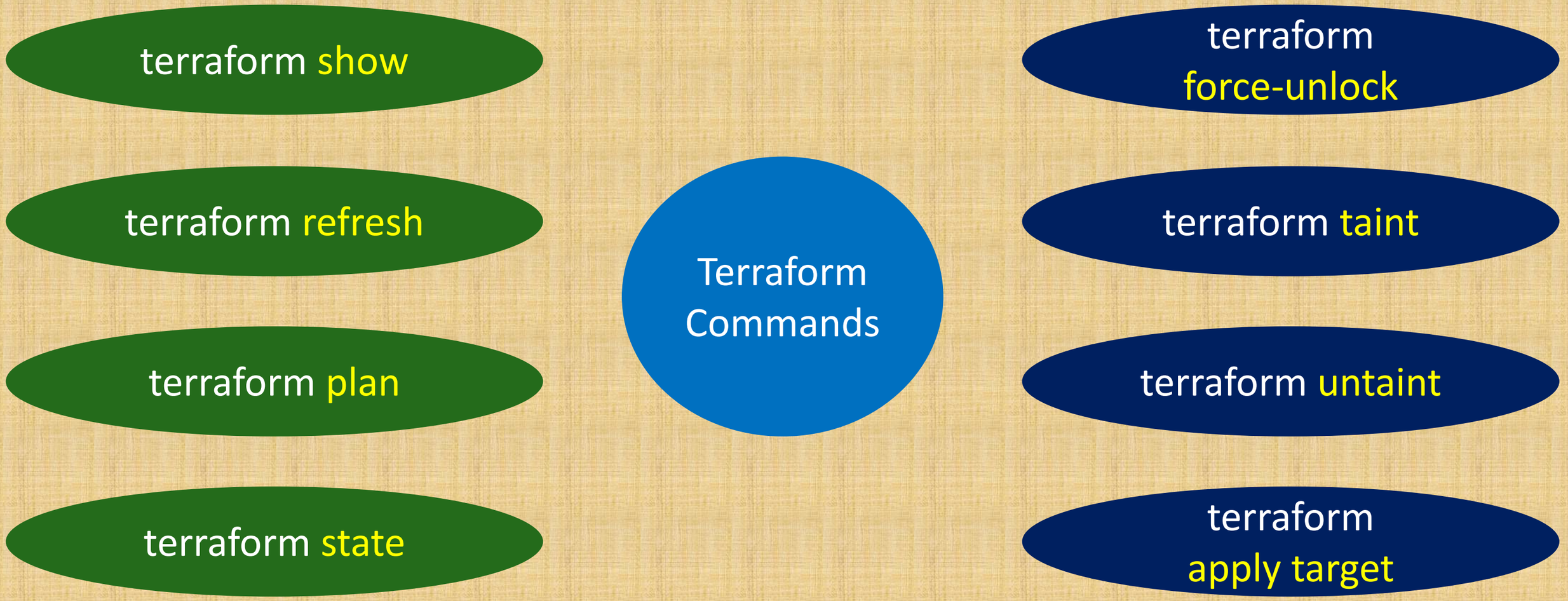
Terraform State Locking

```
# Terraform Block
terraform {
  required_version = "~> 0.14" # which means any ve
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
  # Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket = "terraform-stacksimplify"
    key    = "dev/terraform.tfstate"
    region = "us-east-1"

    # Enable during Step-09
    # For State Locking
    dynamodb_table = "terraform-dev-state-table"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

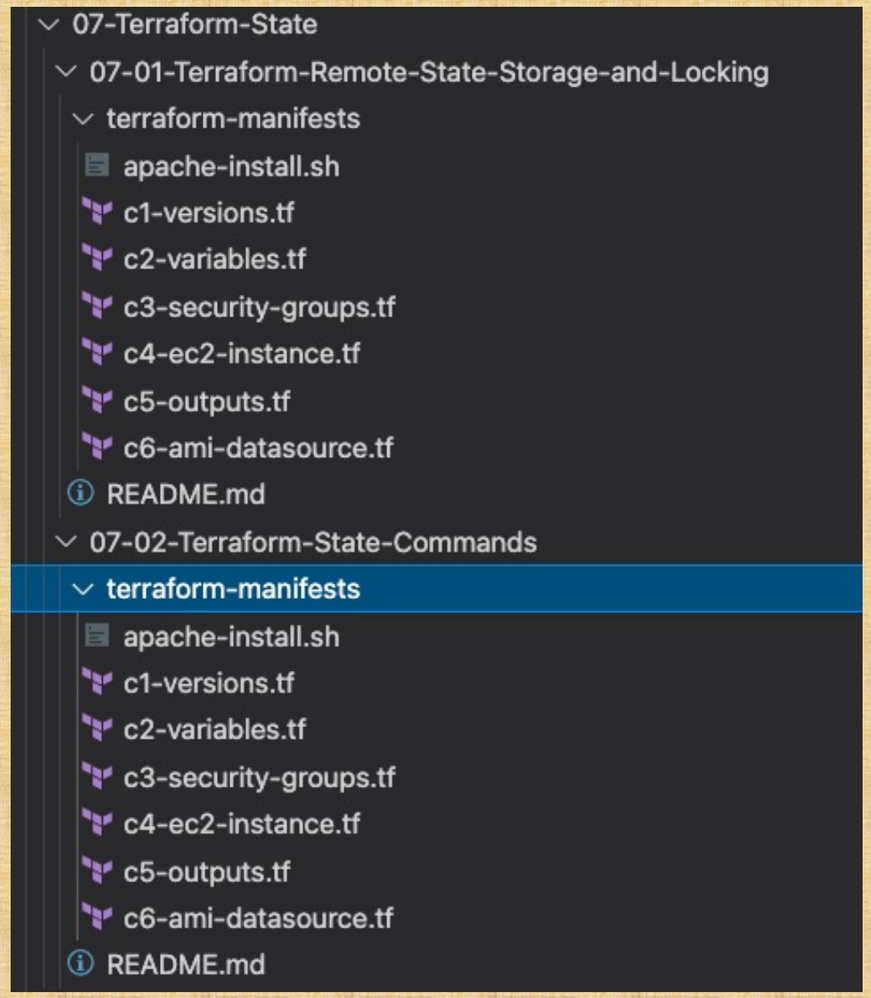
Terraform Commands – State Perspective

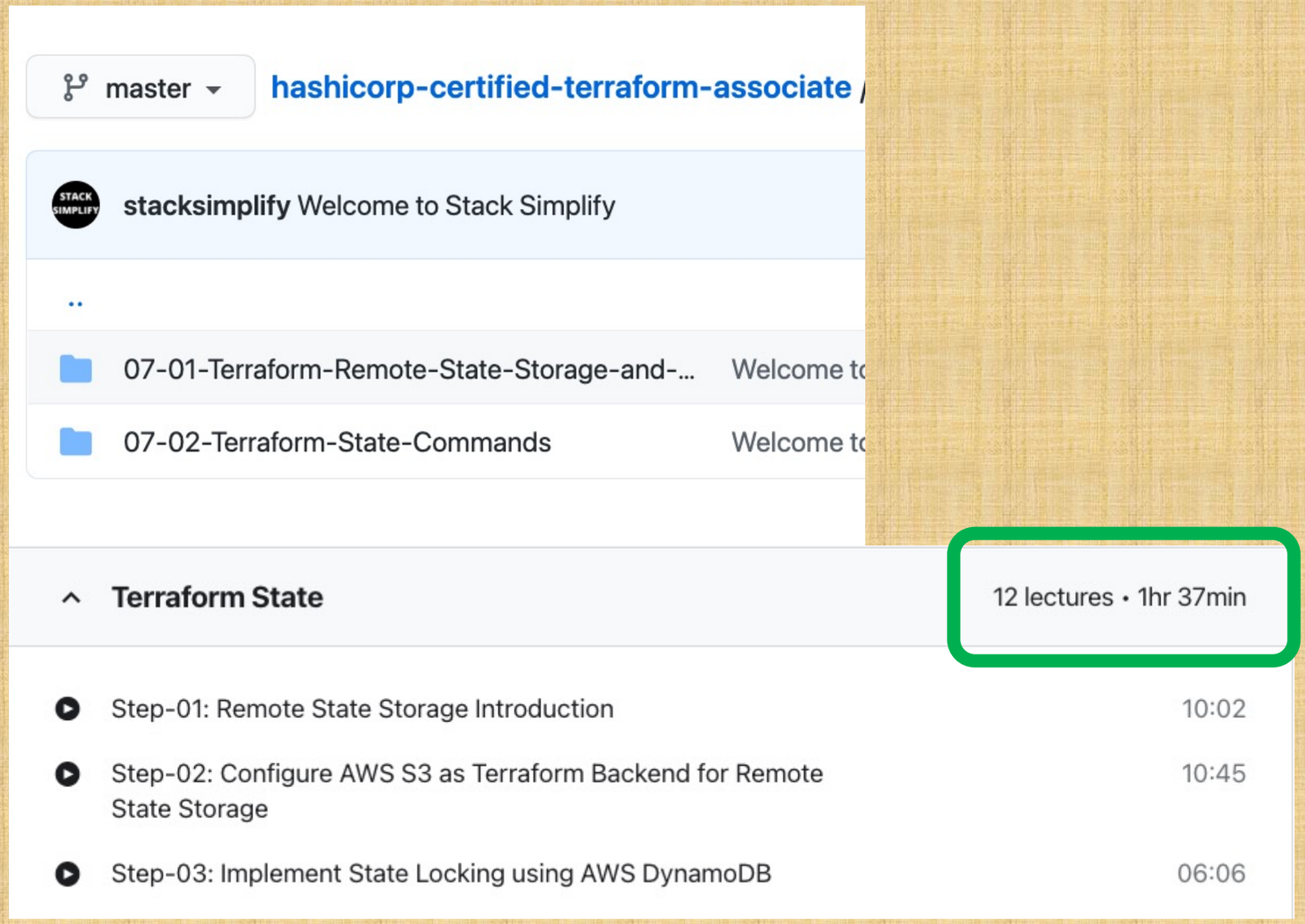


Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples with Step-by-Step Documentation

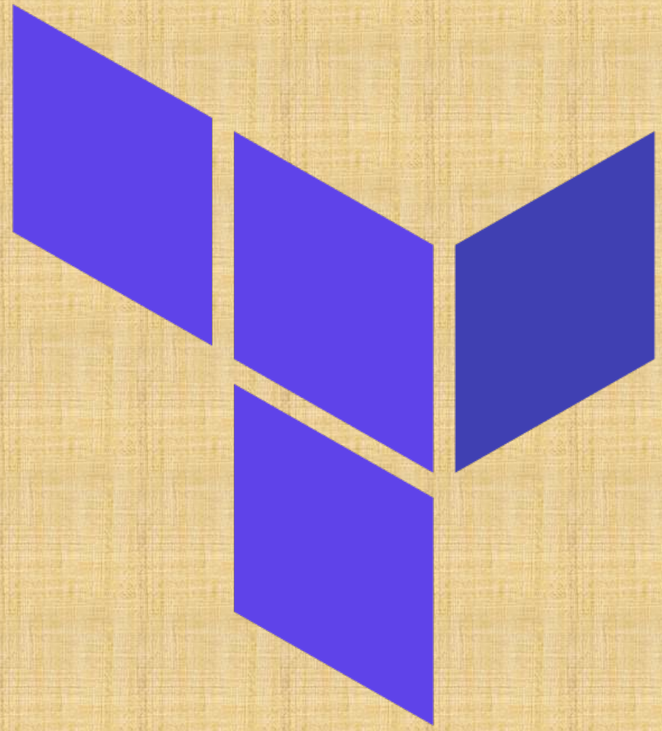




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Backend
Remote State Storage



Author: Nho Luong

Skill: DevOps Engineer Lead

What is Terraform Backend ?

Backends are responsible for storing state and providing an API for state locking.

Terraform
State Storage

AWS S3 Bucket

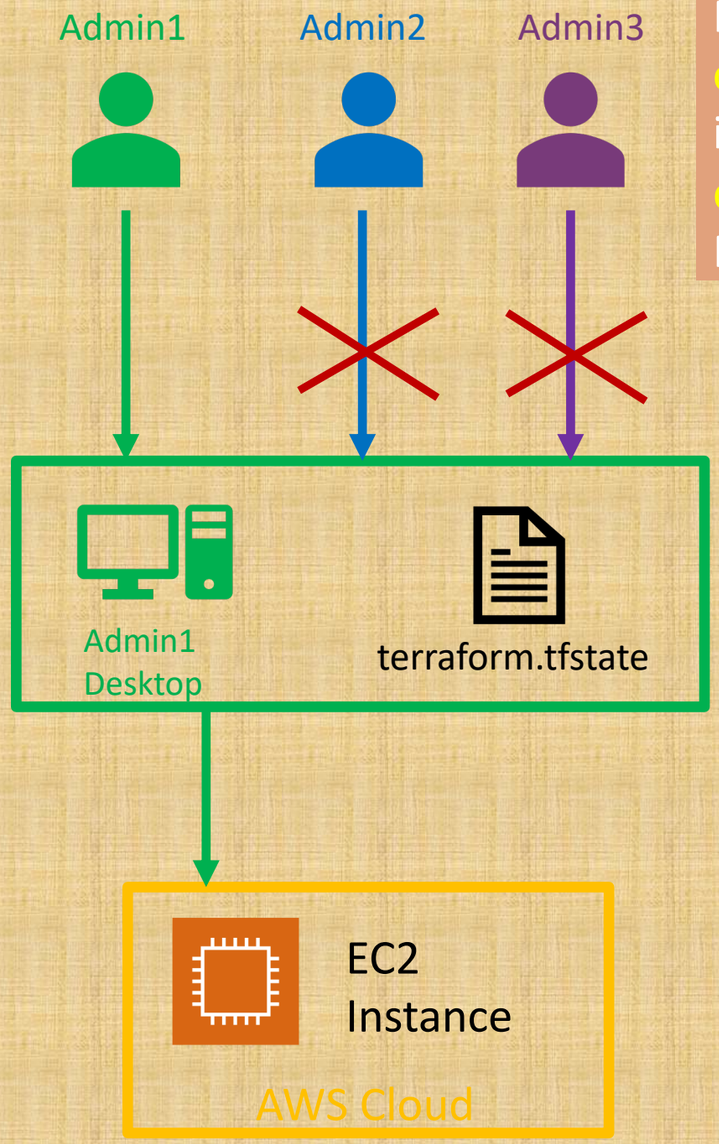
Terraform
State Locking

AWS DynamoDB

Author: Nho Luong

Skill: DevOps Engineer Lead

Local State File



Multiple Team members cannot update the infrastructure as they don't have access to State File

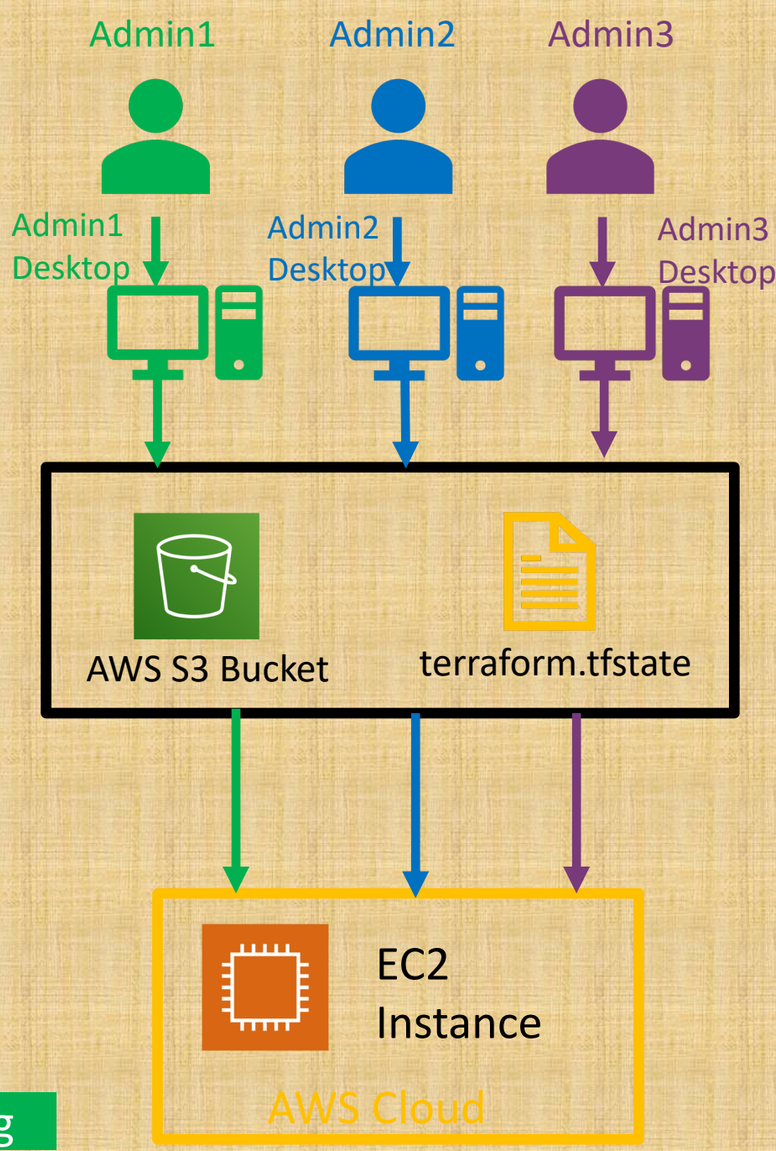
This means we need store the state file in a shared location.

Remote State File

Using Terraform Backend concept we can use AWS S3 as the shared storage for State Files

If two team members are running Terraform at the same time, you may run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.

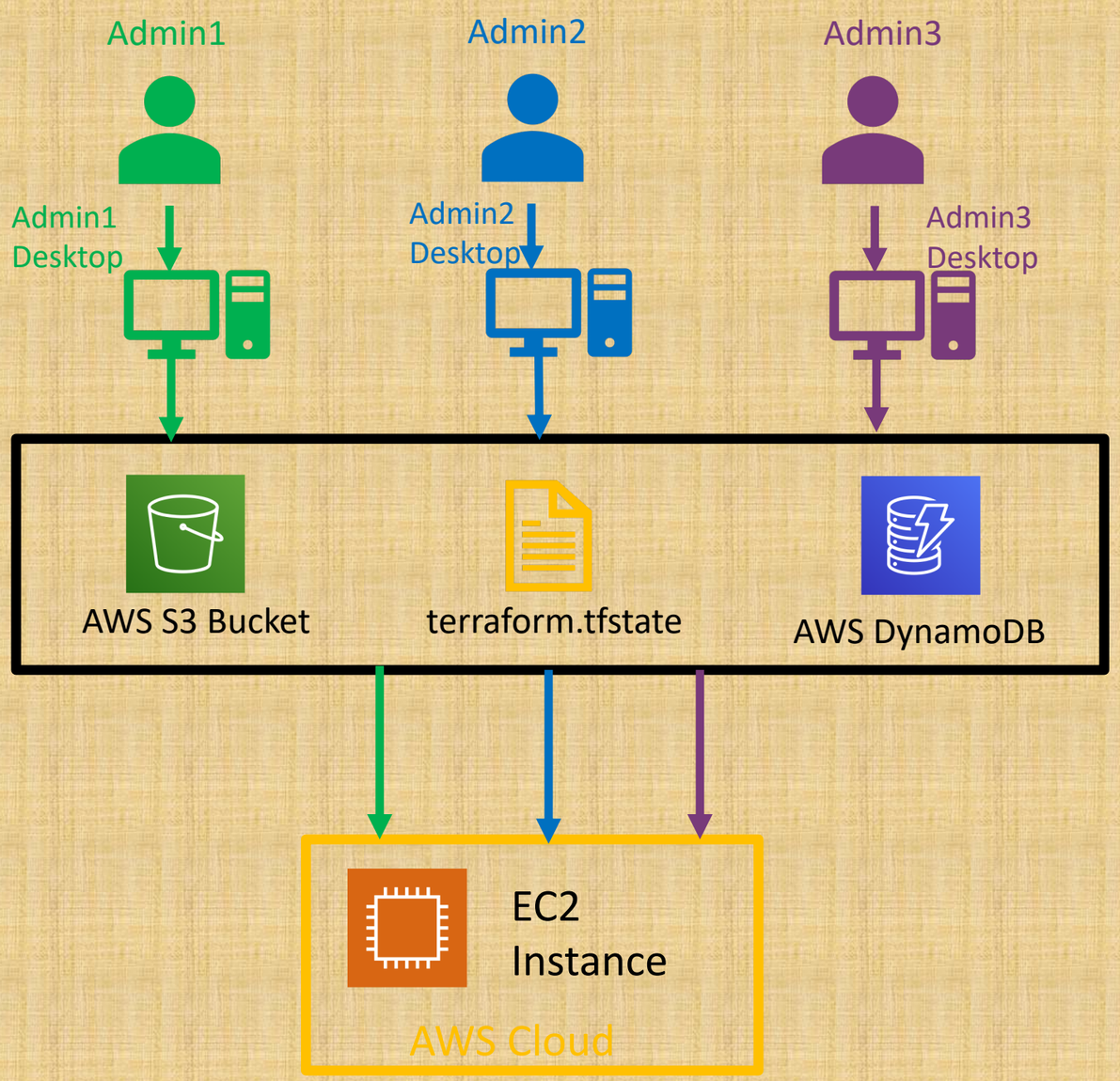
Implement State Locking



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Remote State File with State Locking



Not all backends support State Locking. AWS S3 supports State Locking

State locking happens automatically on all operations that could write state.

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.

If acquiring the lock is taking longer than expected, Terraform will output a status message.

If Terraform doesn't output a message, state locking is still occurring if your backend supports it.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Remote State File with State Locking

Terraform State Storage to Remote Backend

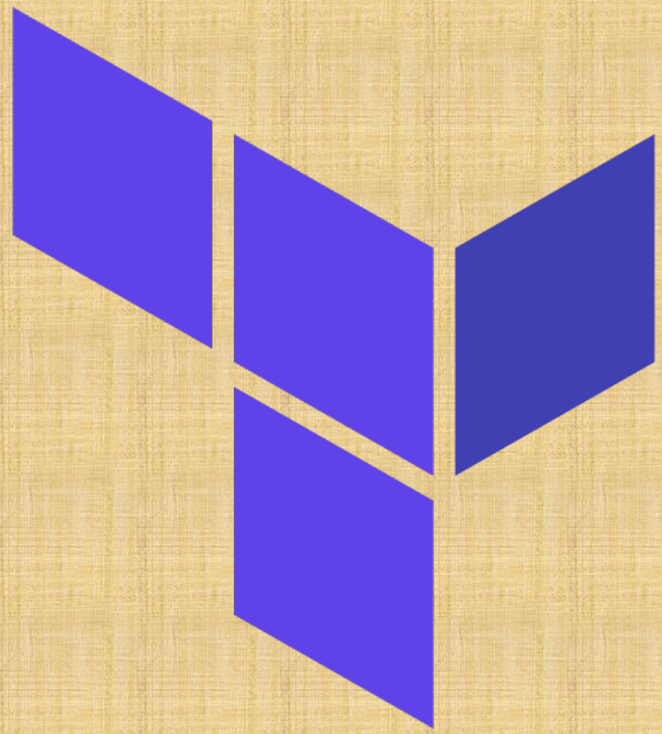
Terraform State Locking

```
# Terraform Block
terraform {
  required_version = "~> 0.14" # which means any ve
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
  # Adding Backend as S3 for Remote State Storage
  backend "s3" {
    bucket = "terraform-stacksimplify"
    key    = "dev/terraform.tfstate"
    region = "us-east-1"

    # Enable during Step-09
    # For State Locking
    dynamodb_table = "terraform-dev-state-table"

  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Backends



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Backends

Each Terraform configuration can specify a backend, which defines where and how operations are performed, where state snapshots are stored, etc.

Where Backends are Used

Backend configuration is only used by Terraform CLI.

Terraform Cloud and Terraform Enterprise always use their own state storage when performing Terraform runs, so they ignore any backend block in the configuration.

For Terraform Cloud users also it is always recommended to use backend block in Terraform configuration for commands like terraform taint which can be executed only using Terraform CLI

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Backends

What Backends Do

There are two things backends will be used for

1. Where state is stored
2. Where operations are performed.

Store State

Terraform uses persistent state data to keep track of the resources it manages.

Everyone working with a given collection of infrastructure resources must be able to access the same state data (shared state storage).

State Locking

State Locking is to prevent conflicts and inconsistencies when the operations are being performed

What are Operations ?
terraform apply
terraform destroy

Operations

"Operations" refers to performing API requests against infrastructure services in order to create, read, update, or destroy resources.

Not every terraform subcommand performs API operations; many of them only operate on state data.

Only two backends actually perform operations: local and remote.

The remote backend can perform API operations remotely, using Terraform Cloud or Terraform Enterprise.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Backends

Backend Types

Enhanced Backends

Enhanced backends can both store state and perform operations. There are only two enhanced backends: local and remote

Example for Remote Backend Performing Operations : Terraform Cloud, Terraform Enterprise

Standard Backends

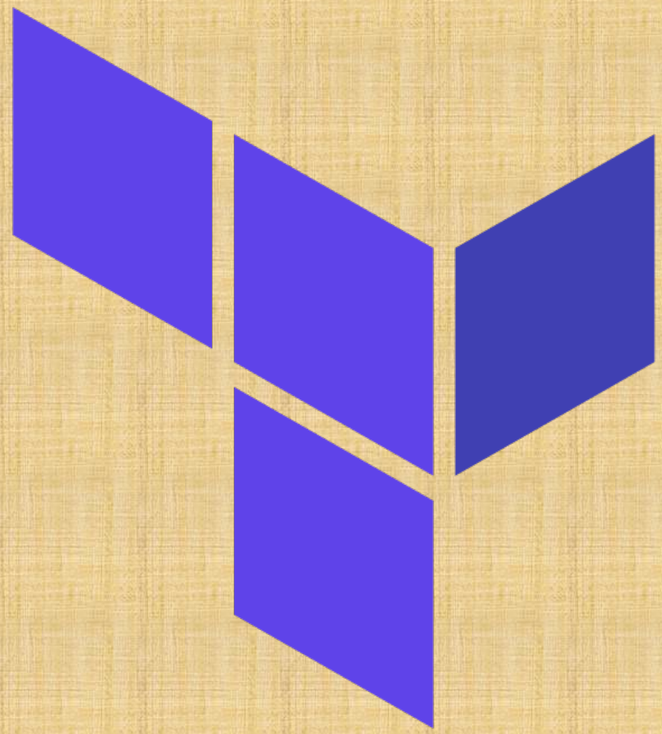
Standard backends only store state, and rely on the local backend for performing operations.

Example: AWS S3, Azure RM, Consul, etcd, gcs http and many more

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Workspaces



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Workspaces – CLI based

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.

By default, we are working in default workspace

Named workspaces allow conveniently switching between multiple instances of a *single* configuration within its *single* backend.

They are convenient in a number of situations, but cannot solve all problems.

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

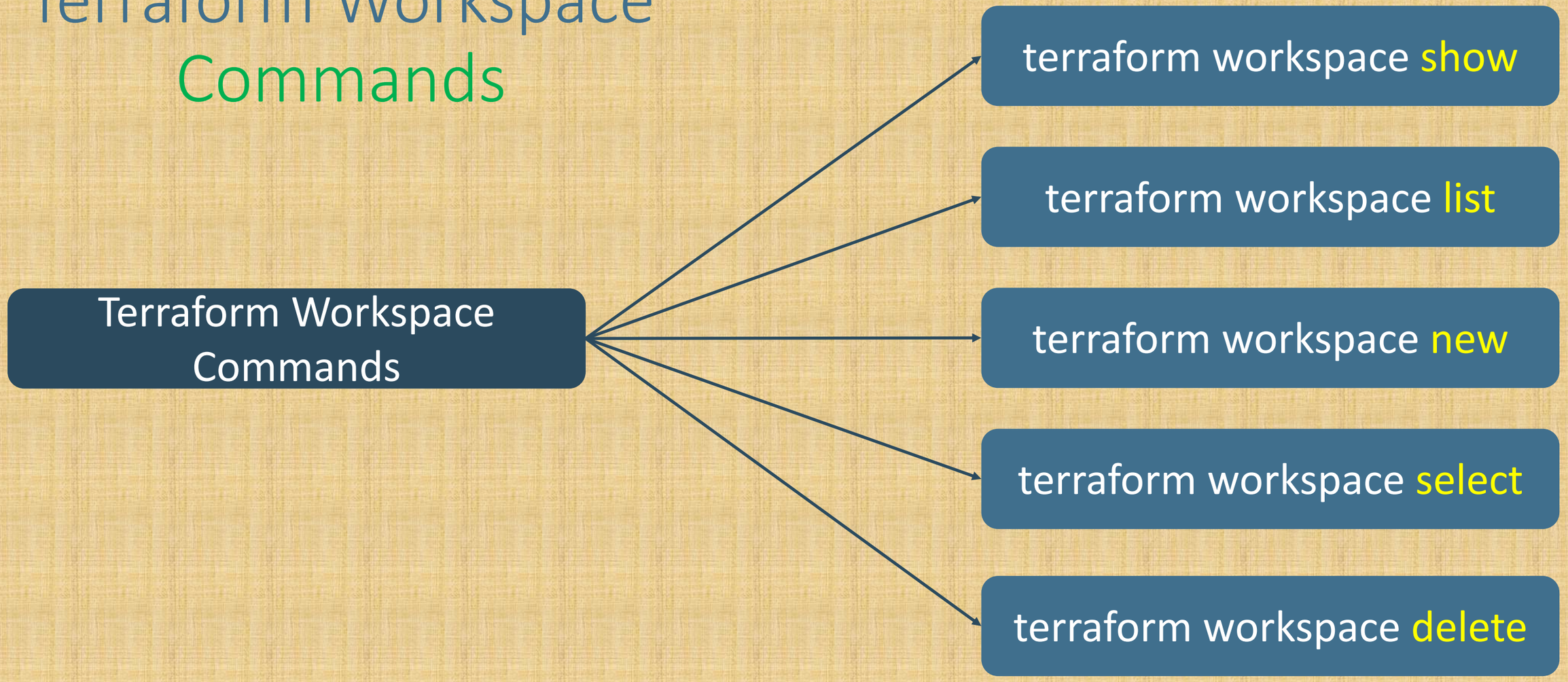
Terraform will not recommend using workspaces for larger infrastructures inline with environments pattern like dev, qa, staging. Recommended to use separate configuration directories

Terraform CLI workspaces are completed different from Terraform Cloud Workspaces

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Workspace Commands



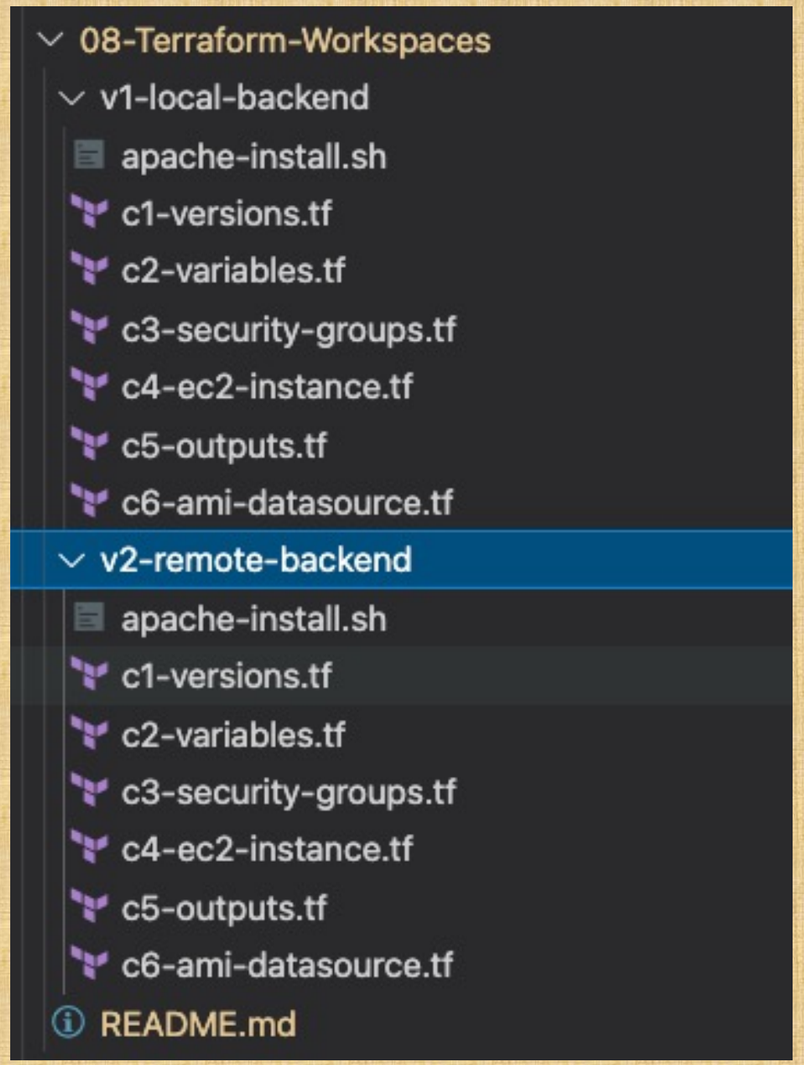
Usecase-1: Local Backend

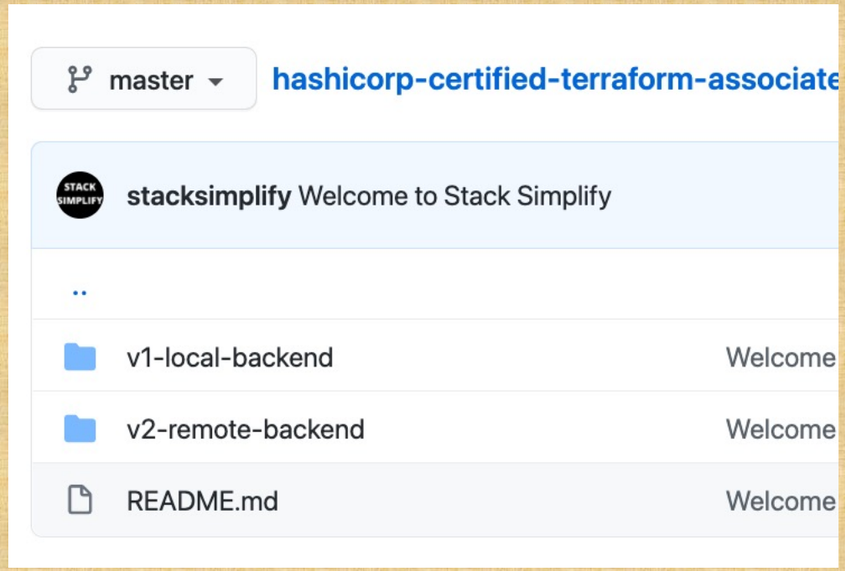
Usecase-2: Remote Backend

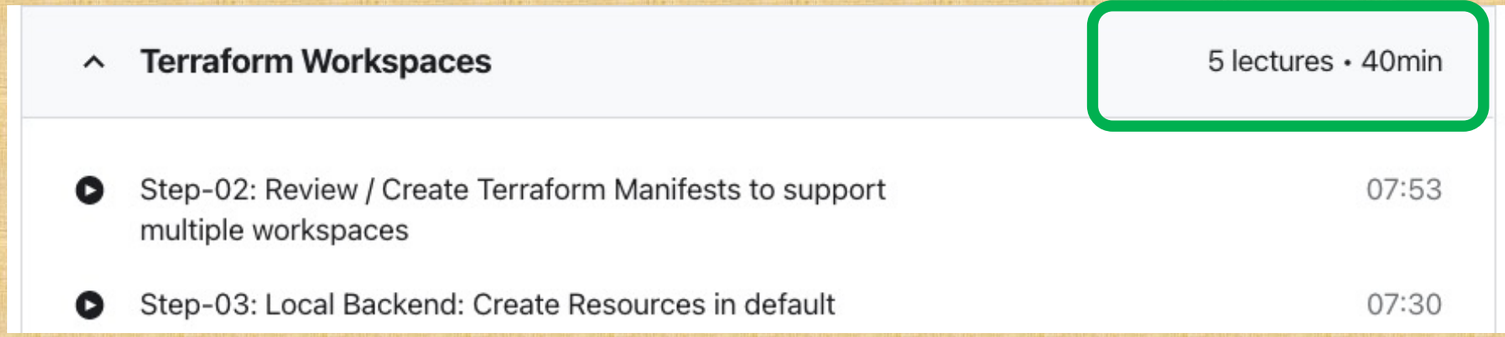
Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Example with Step-by-Step Documentation on Github



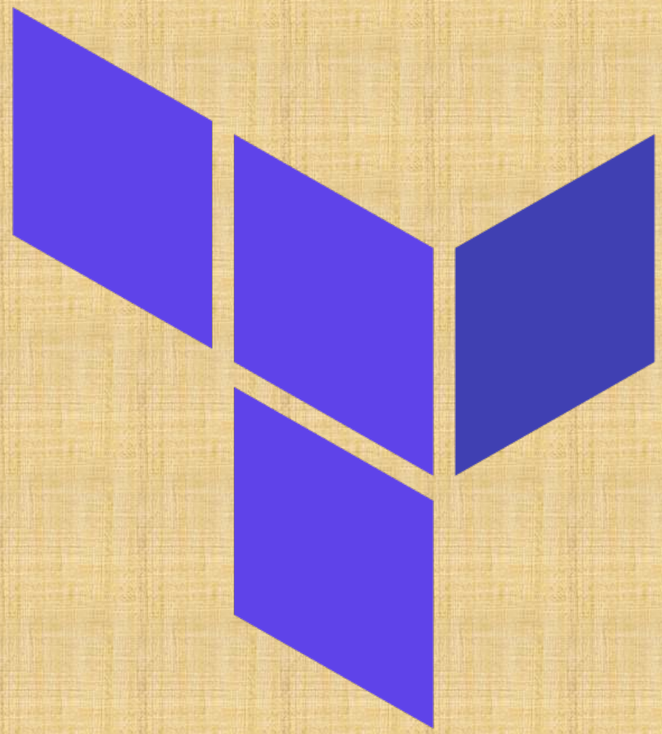




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Provisioners



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Provisioners

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers

Passing data into virtual machines and other compute resources

Running configuration management software (packer, chef, ansible)

Creation-Time Provisioners

Failure Behaviour: Continue: Ignore the error and continue with creation or destruction.

Provisioners are a Last Resort

First-class Terraform provider functionality may be available

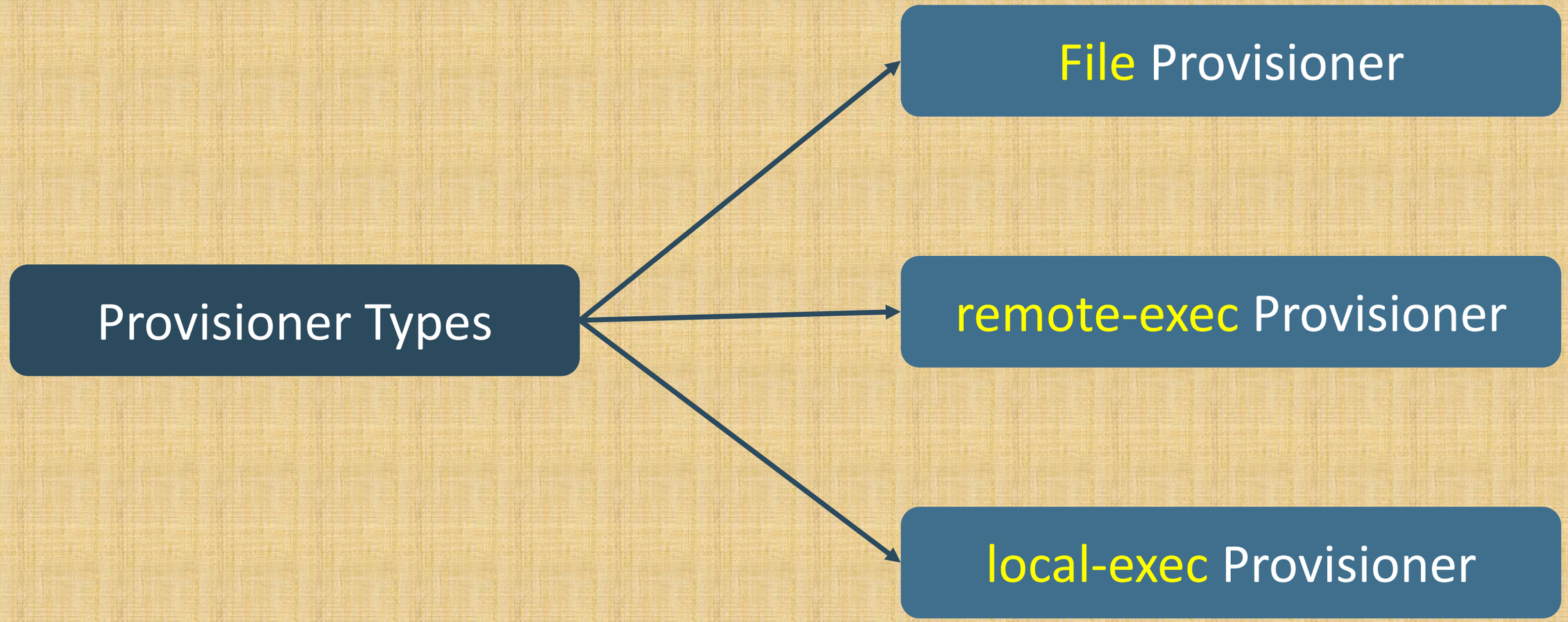
Destroy-Time Provisioners

Failure Behaviour: Fail: Raise an error and stop applying (the default behavior). If creation provisioner, taint resource

Author: Nho Luong

Skill: DevOps Engineer Lead

Types of Provisioners



Author: Nho Luong

Skill: DevOps Engineer Lead

Connection Block

Most provisioners require access to the remote resource via SSH or WinRM, and expect a nested connection block with details about how to connect.

Expressions in connection blocks cannot refer to their parent resource by name. Instead, they can use the special self object.

```
# Connection Block for Provisioners to connect to EC2
connection {
  type = "ssh"
  host = self.public_ip # Understand what is "self"
  user = "ec2-user"
  password = ""
  private_key = file("private-key/terraform-key.pem")
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

File Provisioner

File Provisioner

- File Provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource.
- The file provisioner supports both ssh and winrm type of connections

> hashicorp-certified-terraform-associate > 09-Terraform-Provisioners > 09-01-File-Provisioner > terraform-manifests

```
# Create EC2 Instance - Amazon2 Linux
resource "aws_instance" "my-ec2-vm" {
  ami           = data.aws_ami.amzlinux.id
  instance_type = var.instance_type
  key_name      = "terraform-key"
  #count = terraform.workspace == "default" ? 1 : 1
  user_data = file("apache-install.sh")
  vpc_security_group_ids = [aws_security_group.vpc-ssh.id,
  tags = {
    "Name" = "vm-${terraform.workspace}-0"
  }
# PLAY WITH /tmp folder in EC2 Instance with File Provisio
  # Connection Block for Provisioners to connect to EC2 In
  connection {
    type = "ssh"
    host = self.public_ip # Understand what is "self"
    user = "ec2-user"
    password = ""
    private_key = file("private-key/terraform-key.pem")
  }
```

```
# Copies the file-copy.html file to /tmp/
 provisioner "file" {
    source      = "apps/file-copy.html"
    destination = "/tmp/file-copy.html"
 }

 # Copies the string in content into /tmp
 provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
 }

 # Copies the app1 folder to /tmp - FOLDE
 provisioner "file" {
    source      = "apps/app1"
    destination = "/tmp"
 }
```

Author: Nho Luong

Skill: DevOps Engineer Lead

local-exec Provisioner

local-exec Provisioner

- The local-exec provisioner invokes a local executable after a resource is created.
- This invokes a process on the machine running Terraform, not on the resource.

```
# local-exec provisioner (Creation-Time Provisioner - Triggered during Create Resource)
provisioner "local-exec" {
  command = "echo ${aws_instance.my-ec2-vm.private_ip} >> creation-time-private-ip.txt"
  working_dir = "local-exec-output-files/"
  #on_failure = continue
}

# local-exec provisioner - (Destroy-Time Provisioner - Triggered during Destroy Resource)
provisioner "local-exec" {
  when    = destroy
  command = "echo Destroy-time provisioner Instanace Destroyed at `date` >> destroy-time.txt"
  working_dir = "local-exec-output-files/"
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

remote-exec Provisioner

remote-exec Provisioner

- The remote-exec provisioner invokes a script on a remote resource after it is created.
- This can be used to run a configuration management tool, bootstrap into a cluster, etc.

```
# Copies the file-copy.html file to /tmp/file-copy.html
  provisioner "file" {
    source      = "apps/file-copy.html"
    destination = "/tmp/file-copy.html"
  }

# Copies the file to Apache Webserver /var/www/html directory
  provisioner "remote-exec" {
    inline = [
      "sleep 120",  # Will sleep for 120 seconds to ensure Apache v
      "sudo cp /tmp/file-copy.html /var/www/html"
    ]
  }
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Null-Resource & Provisioners

null_resource

- If you need to run provisioners that aren't directly associated with a specific resource, you can associate them with a null_resource.
- Instances of null_resource are treated like normal resources, but they don't do anything.
- Same as other resource, you can configure provisioners and connection details on a null_resource.

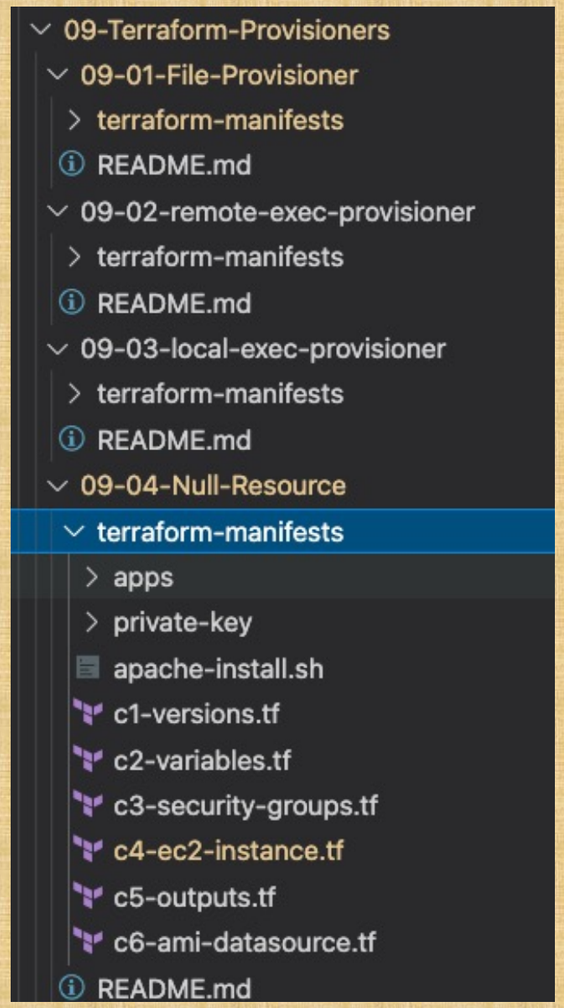
```
# Wait for 90 seconds after creating the above
resource "time_sleep" "wait_90_seconds" {
  depends_on = [aws_instance.my-ec2-vm]
  create_duration = "90s"
}

# Sync App1 Static Content to Webserver using P
resource "null_resource" "sync_app1_static" {
  depends_on = [ time_sleep.wait_90_seconds ]
  triggers = {
    always-update =  timestamp()
  }
```

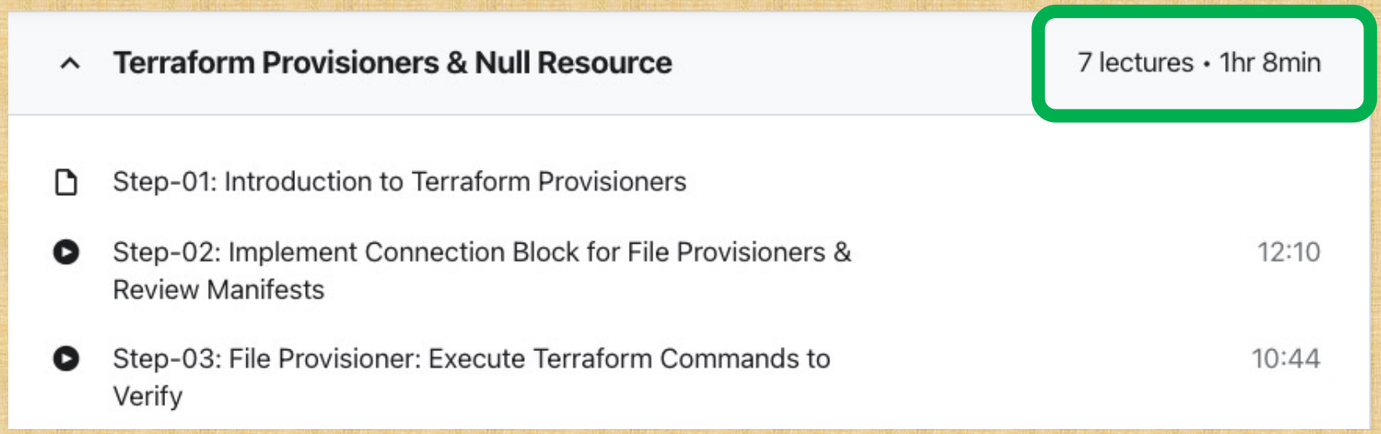
```
 # Connection Block for Provisioners to connect to EC2 I
 connection {
   type = "ssh"
   host = aws_instance.my-ec2-vm.public_ip
   user = "ec2-user"
   password = ""
   private_key = file("private-key/terraform-key.pem")
 }

# Copies the app1 folder to /tmp
 provisioner "file" {
   source      = "apps/app1"
   destination = "/tmp"
 }

# Copies the /tmp/app1 folder to Apache Webserver /var/ww
 provisioner "remote-exec" {
   inline = [
     "sudo cp -r /tmp/app1 /var/www/html"
   ]
 }
```

Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples & Step-by-Step Documentation on Github

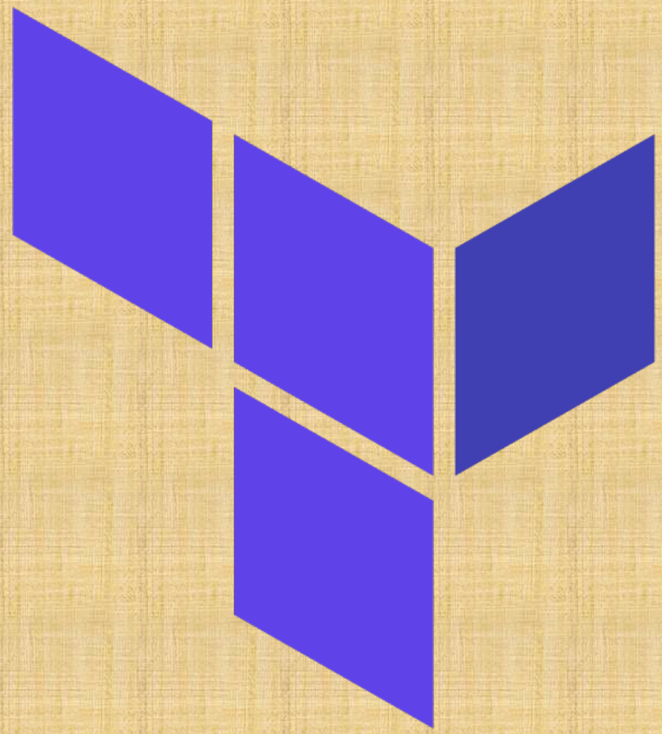




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Modules



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Modules

Modules are containers for multiple resources that are used together. A module consists of a collection of .tf files kept together in a directory.

Modules are the main way to package and reuse resource configurations with Terraform.

A module that has been called by another module is often referred to as a child module.

Every Terraform configuration has at least one module, known as its root module, which consists of the resources defined in the .tf files in the main working directory

Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration.

In addition to modules from the local filesystem, Terraform can load modules from a public or private registry.

This makes it possible to publish modules for others to use, and to use modules that others have published.

Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Modules

Terraform Registry – Publicly Available Modules

The Terraform Registry hosts a broad collection of publicly available Terraform modules for configuring many kinds of common infrastructure.

Demo 1

These modules are free to use, and Terraform can download them automatically if you specify the appropriate source and version in a module call block.

Private Module Registry in Terraform Cloud & Enterprise

Members of your organization might produce modules specifically crafted for your own infrastructure needs.

Demo 2,3

Terraform Cloud and Terraform Enterprise both include a private module registry for sharing modules internally within your organization.

Author: Nho Luong

Skill: DevOps Engineer Lead

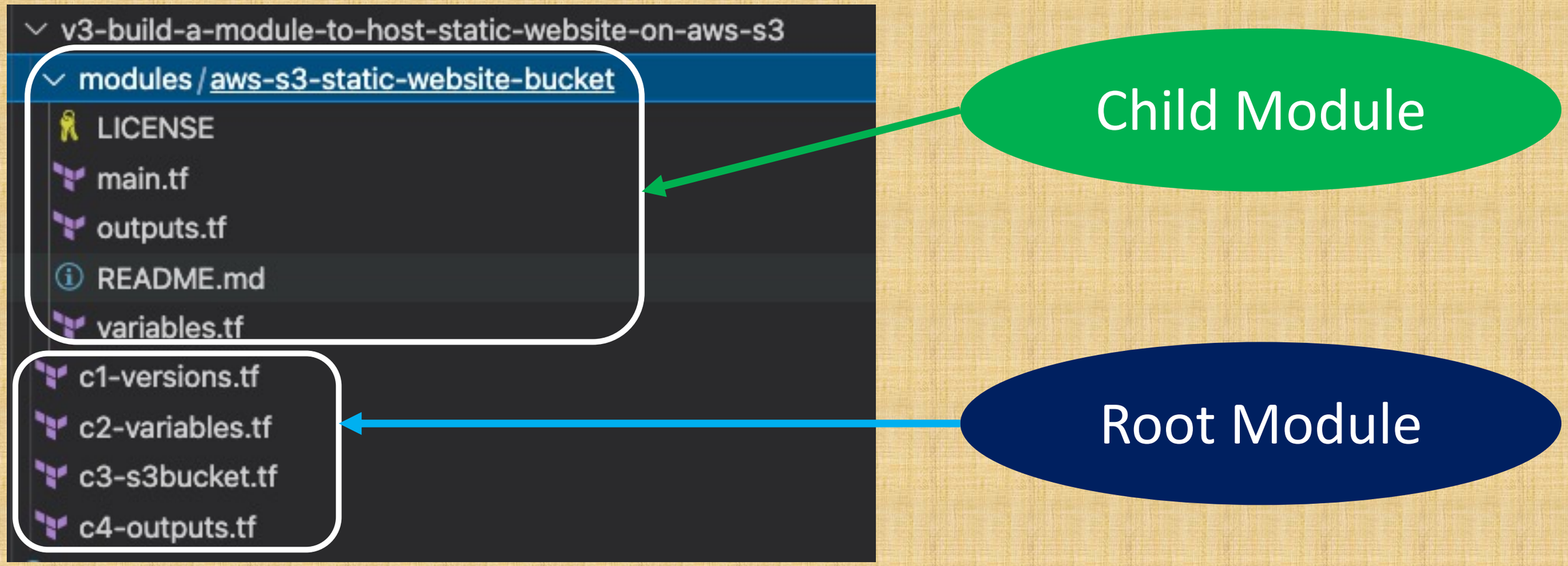
Modules - Demo-1

```
# AWS EC2 Instance Module
module "ec2_cluster" {
  source                 = "terraform-aws-modules/ec2-instance/aws"
  version                = "~> 2.0"

  name                   = "my-modules-demo"
  instance_count         = 2

  ami                    = data.aws_ami.amzlinux.id
  instance_type          = "t2.micro"
  key_name               = "terraform-key"
  monitoring             = true
  vpc_security_group_ids = ["sg-b8406afc"] # Get Default VPC Securit
  subnet_id              = "subnet-4ee95470" # Get one public subnet
  user_data              = file("apache-install.sh")

  tags = {
    Name        = "Modules-Demo"
    Terraform   = "true"
    Environment = "dev"
  }
}
```

Author: Nho Luong

Skill: DevOps Engineer Lead

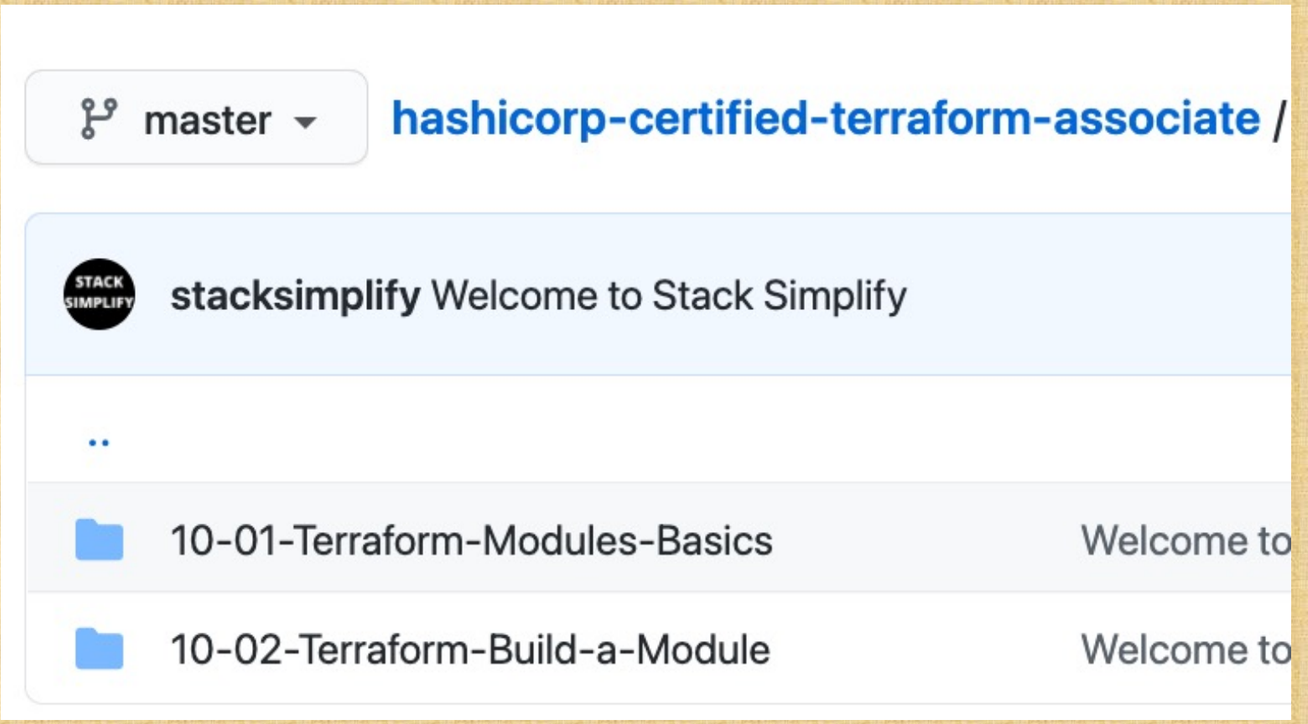
Modules - Demo-2

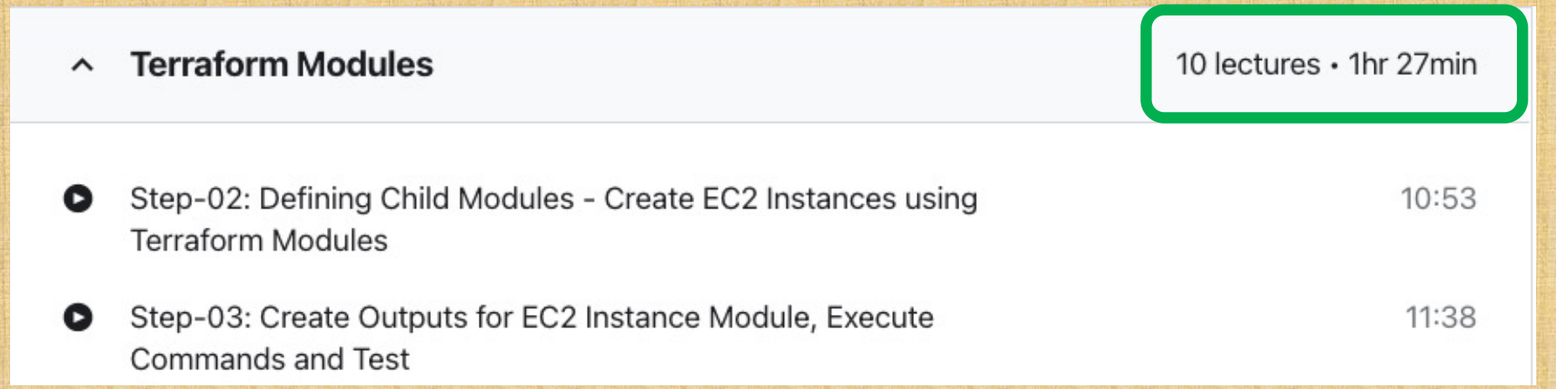


Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples & Step-by-Step Documentation on Github

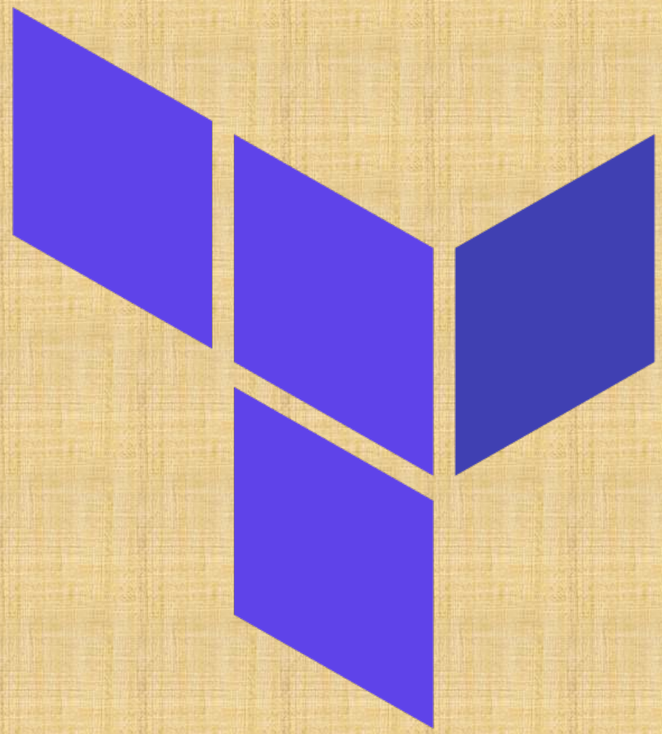




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud
Or
Terraform Enterprise

Terraform Cloud and Terraform Enterprise are different distributions of the same application

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,

Terraform Cloud is available as a hosted service at https://app.terraform.io.

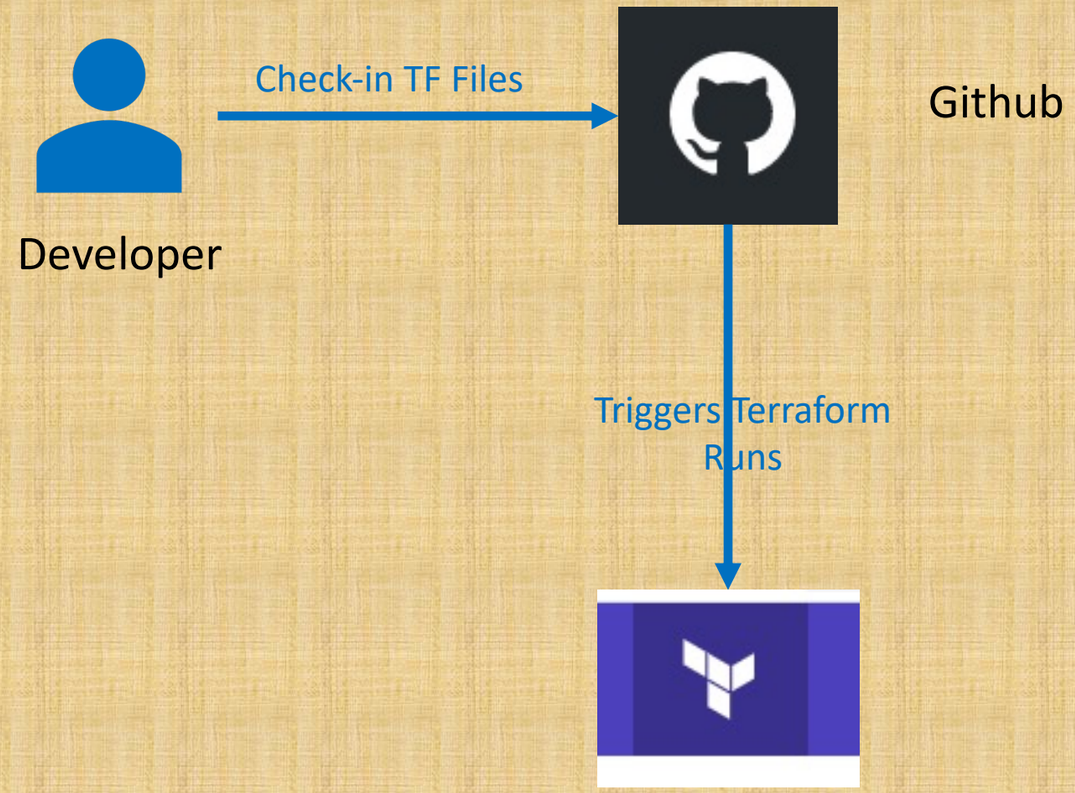
They offer free accounts for small teams, and paid plans with additional feature sets for medium-sized businesses.

Large enterprises can purchase Terraform Enterprise, their self-hosted distribution of Terraform Cloud.

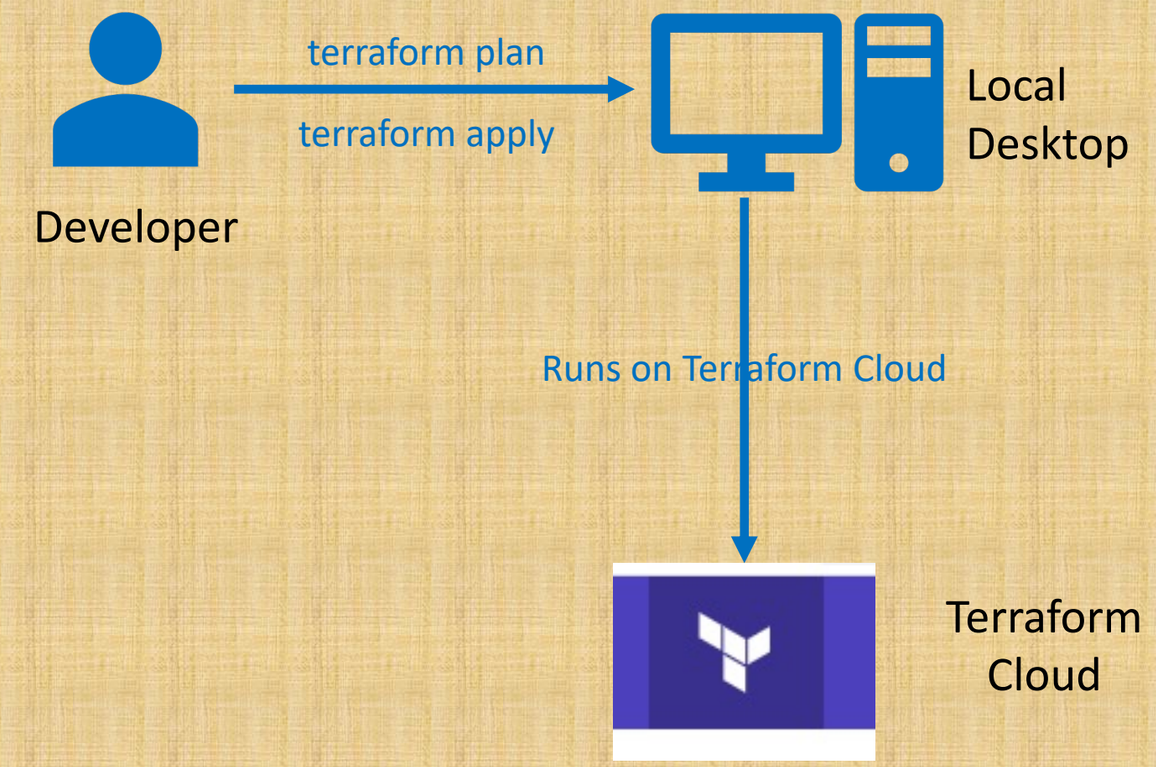
Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform VCS-Driven Workflow



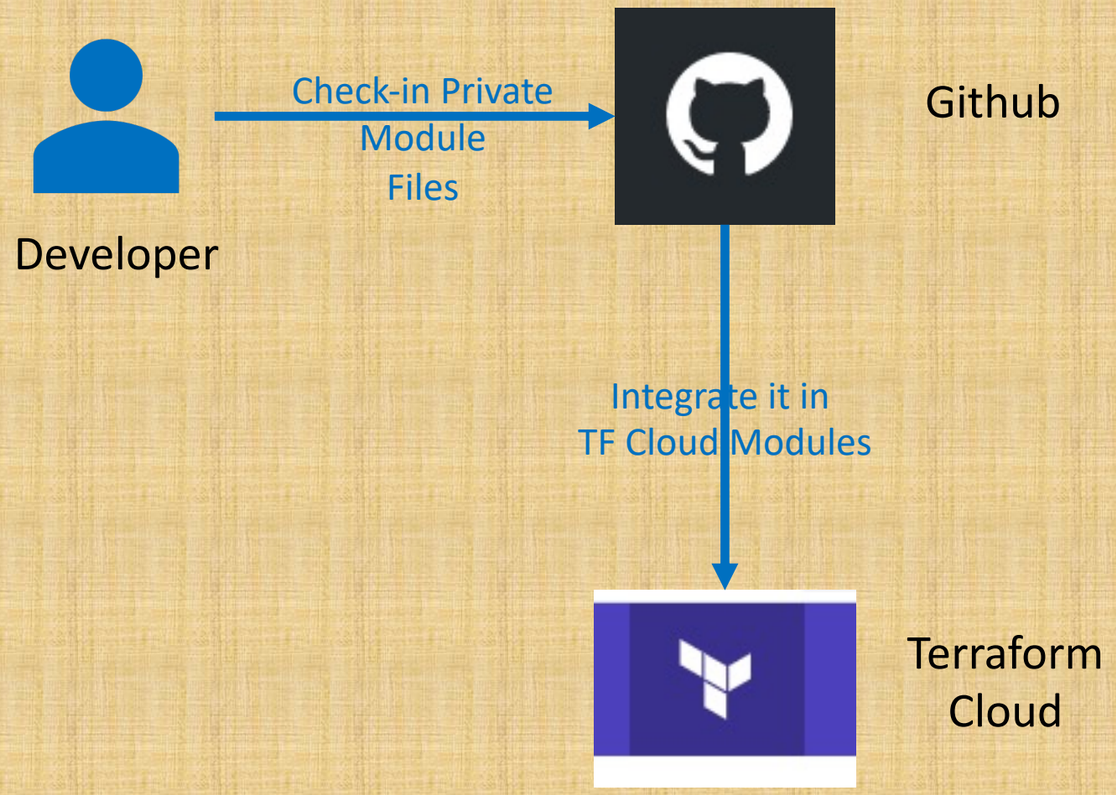
Terraform CLI-Driven Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

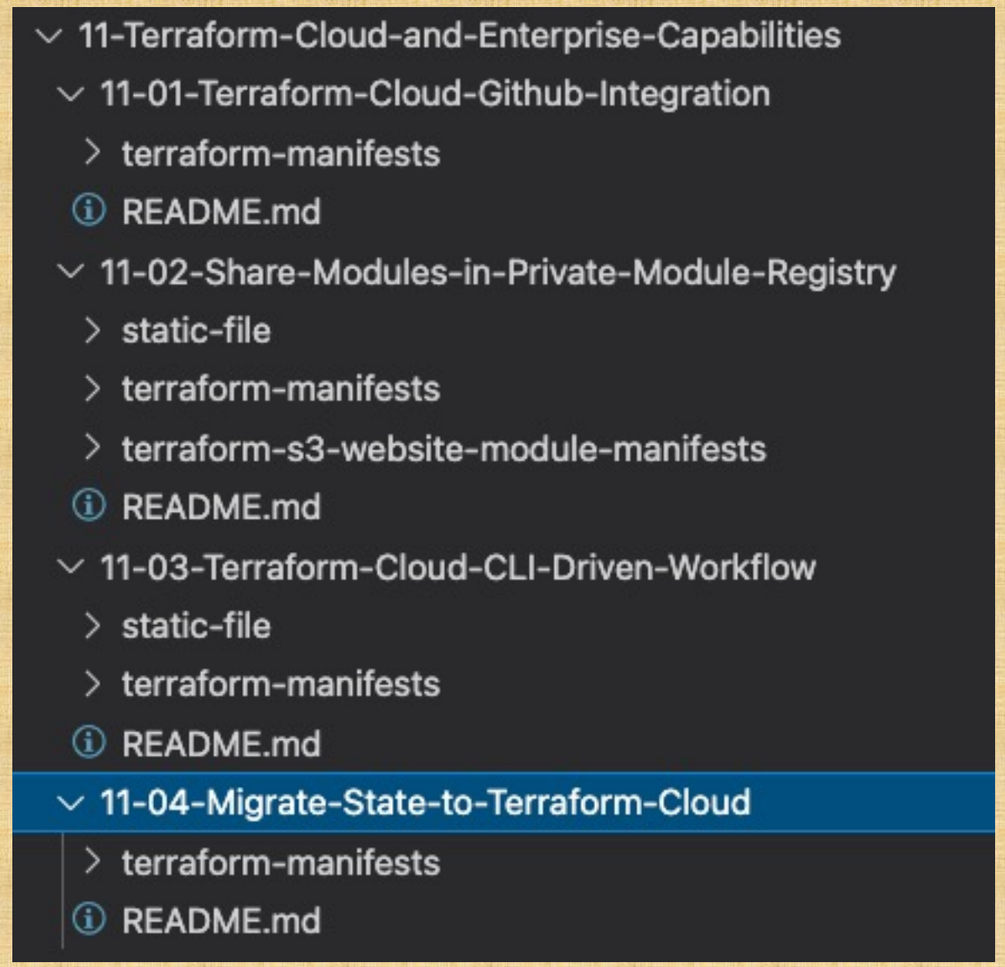
Publish Private Module Registry in Terraform Cloud

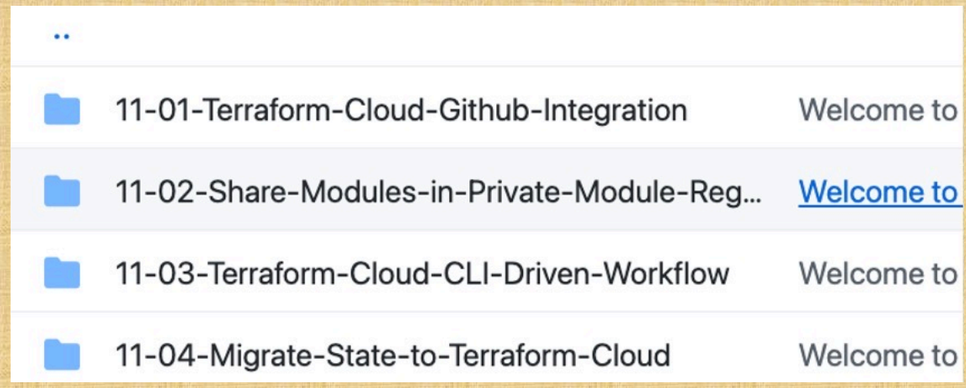


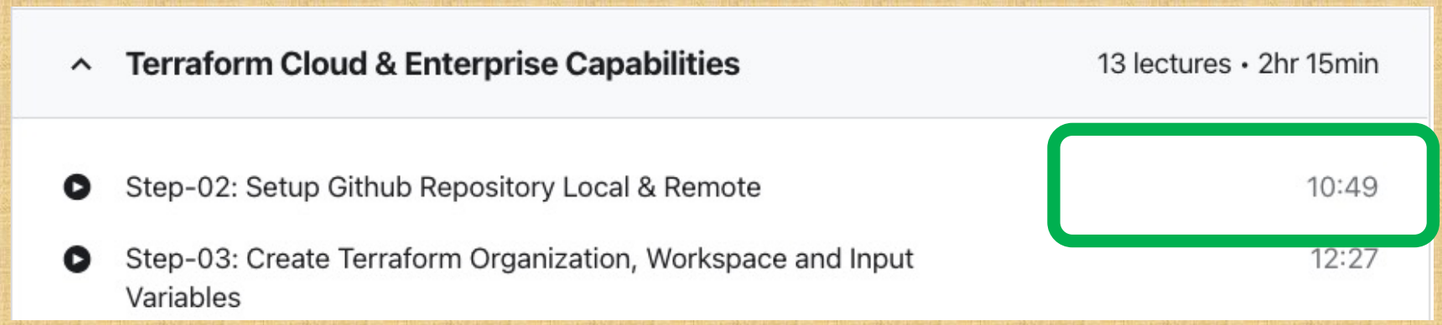
Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples with Step-by-Step Github Documentation



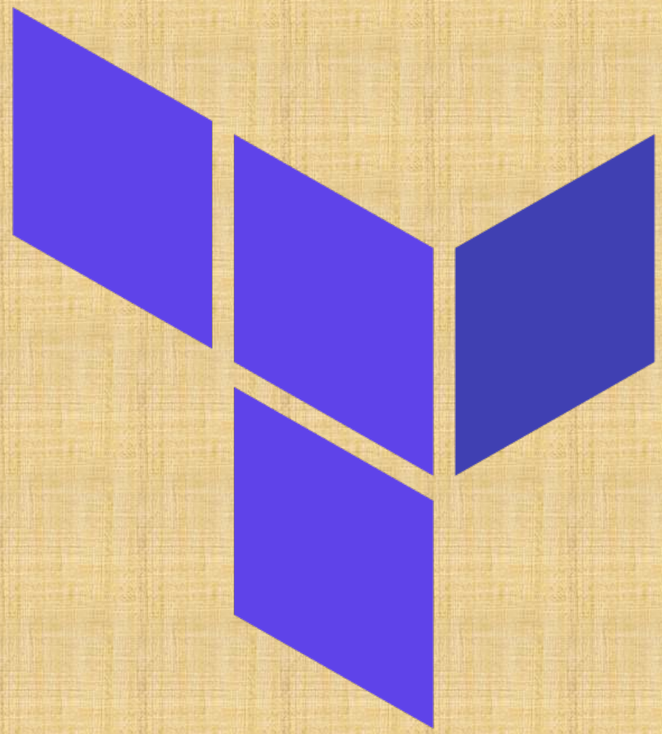




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud Version Control Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

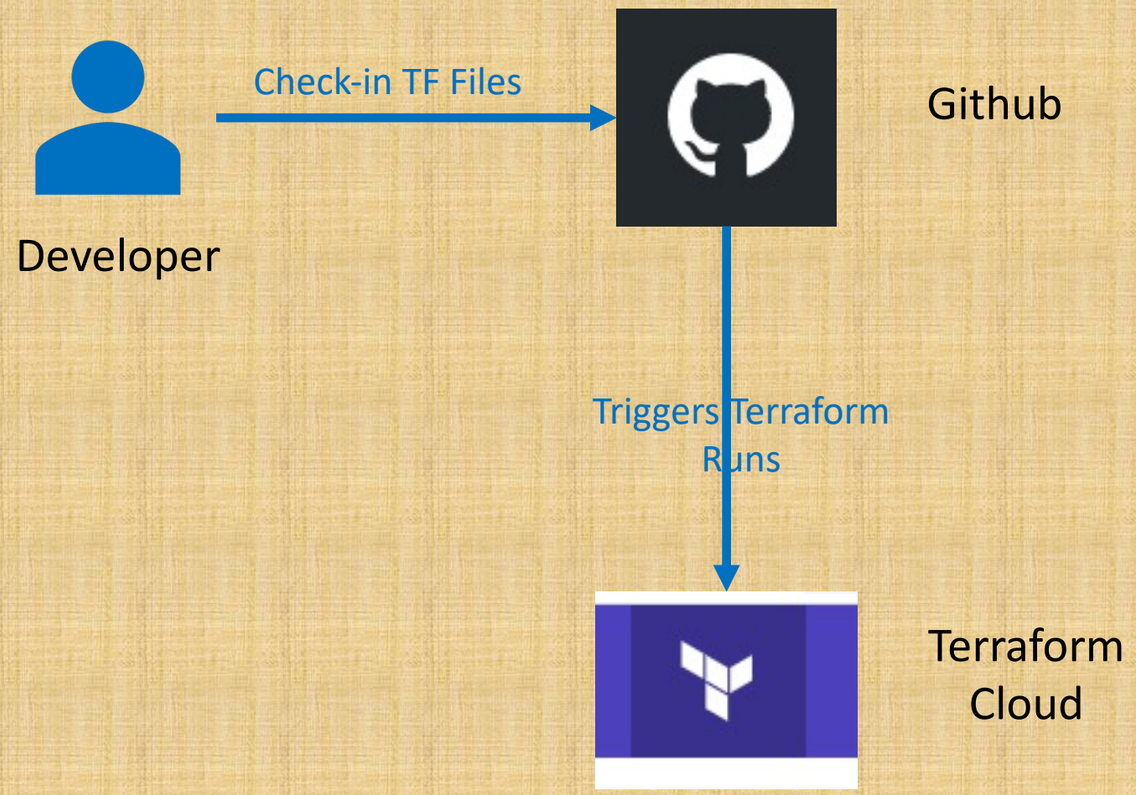
Terraform VCS Integration

Terraform Cloud is more powerful when you integrate it with your version control system (VCS) provider (Github, Bitbucket, Gitlab).

Terraform Cloud can automatically initiate Terraform runs

Terraform Cloud makes code review easier by automatically predicting how pull requests will affect infrastructure.

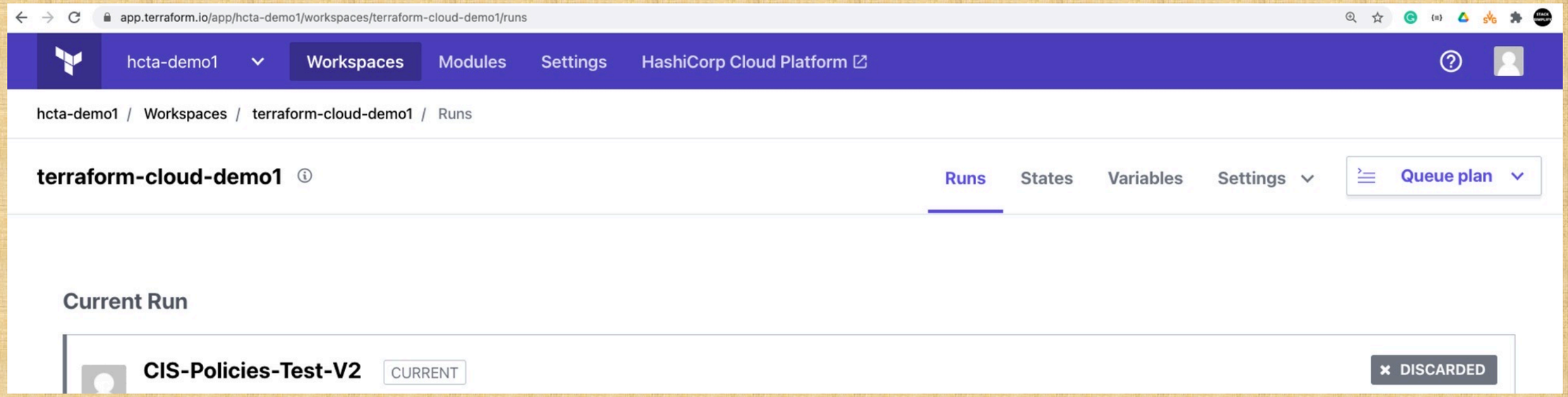
Publishing new versions of a private Terraform module is as easy as pushing a tag to the module's repository.



Author: Nho Luong

Skill: DevOps Engineer Lead

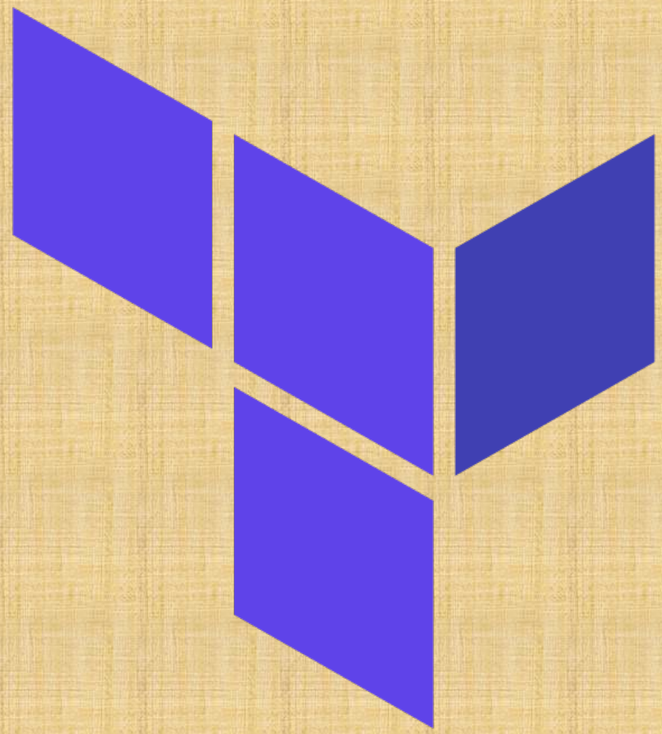
Terraform Cloud - VCS Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud Private Module Registry



Author: Nho Luong

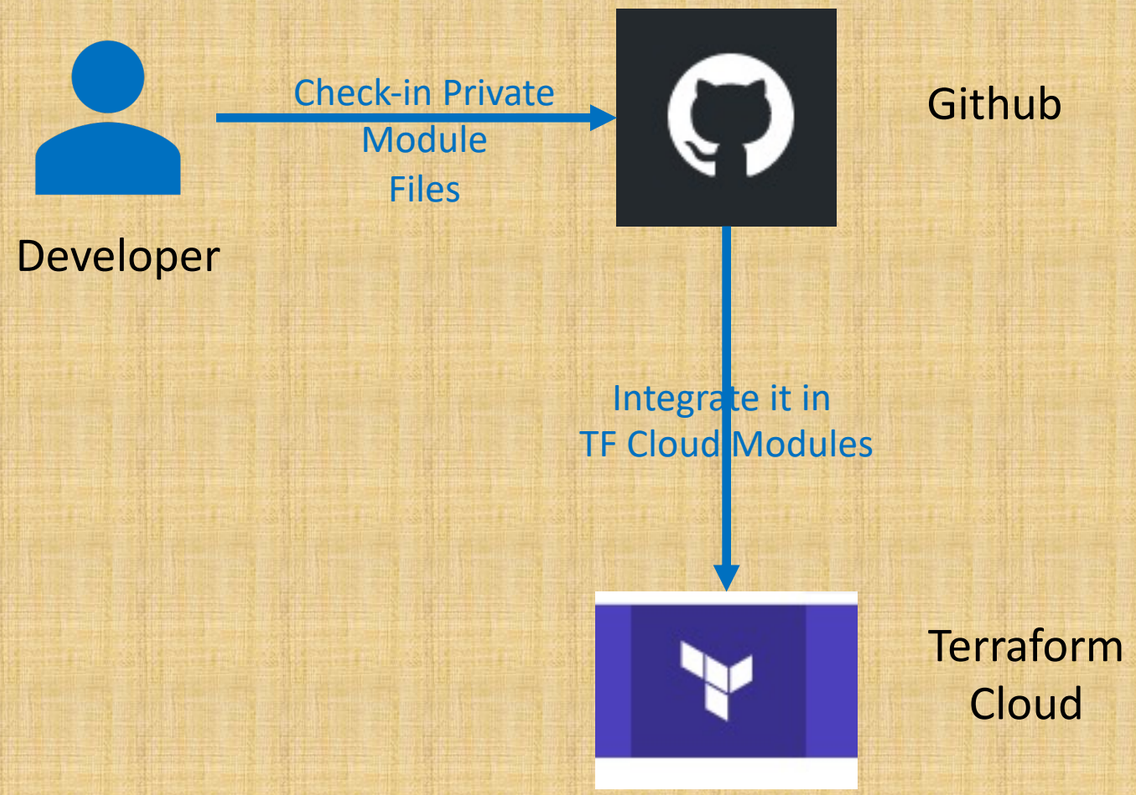
Skill: DevOps Engineer Lead

Terraform Private Module Registry

Terraform Cloud's private module registry helps you share Terraform modules across your organization.

It includes support for module versioning, a searchable and filterable list of available modules, and a configuration designer to help you build new workspaces faster.

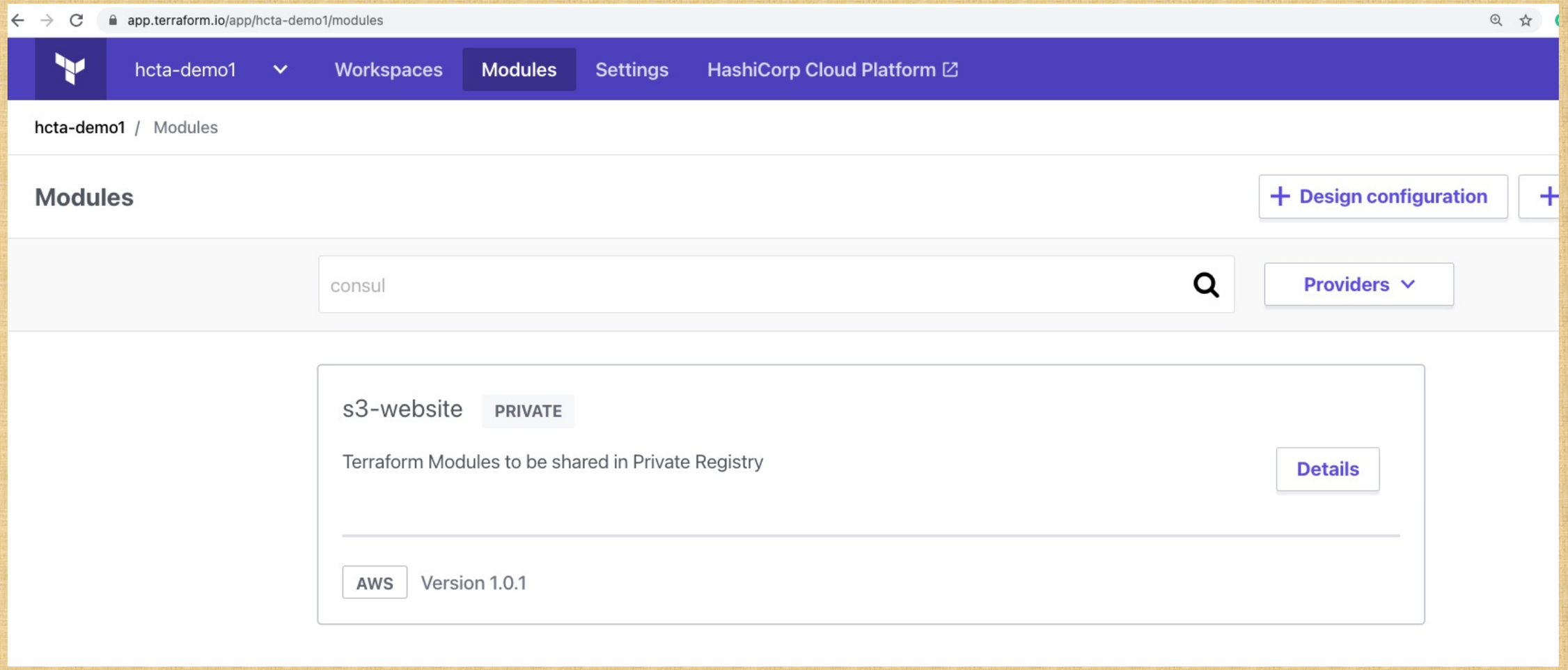
By design, the private module registry works much like the public Terraform Registry. If you're already used the public registry, Terraform Cloud's registry will feel familiar.



Author: Nho Luong

Skill: DevOps Engineer Lead

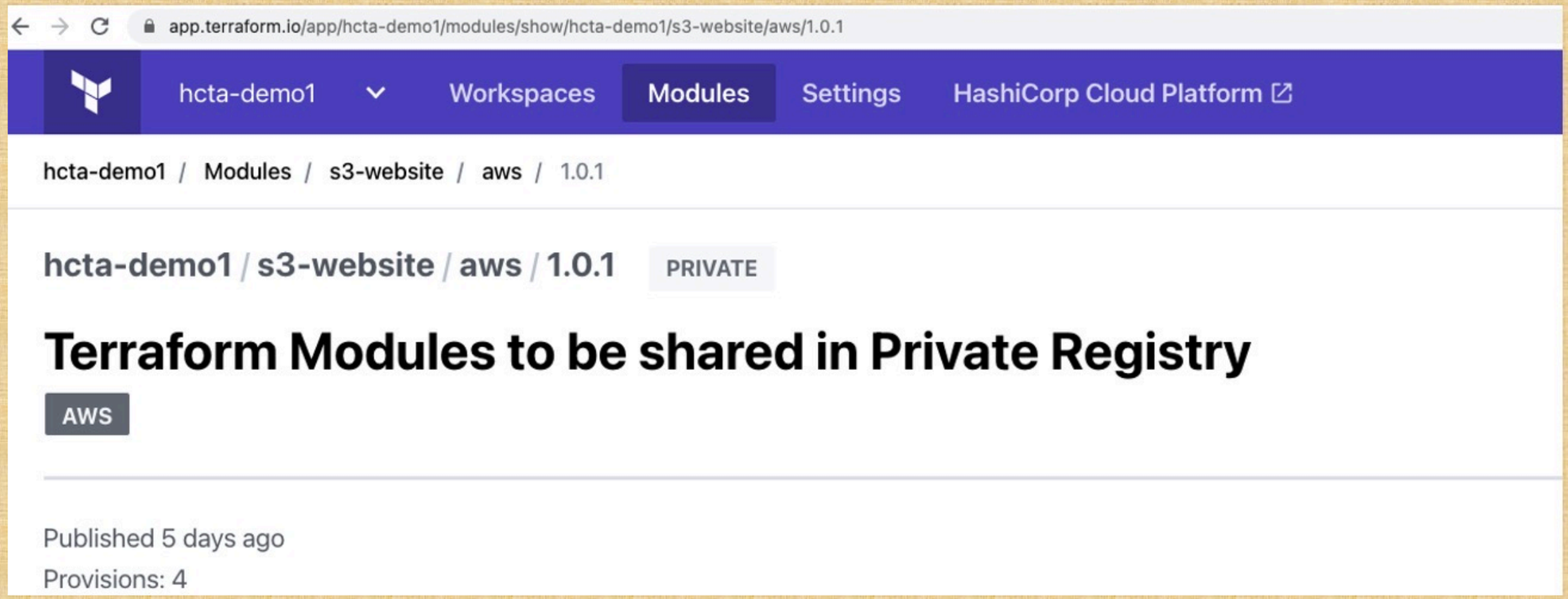
Terraform Cloud – Private Module Registry



Author: Nho Luong

Skill: DevOps Engineer Lead

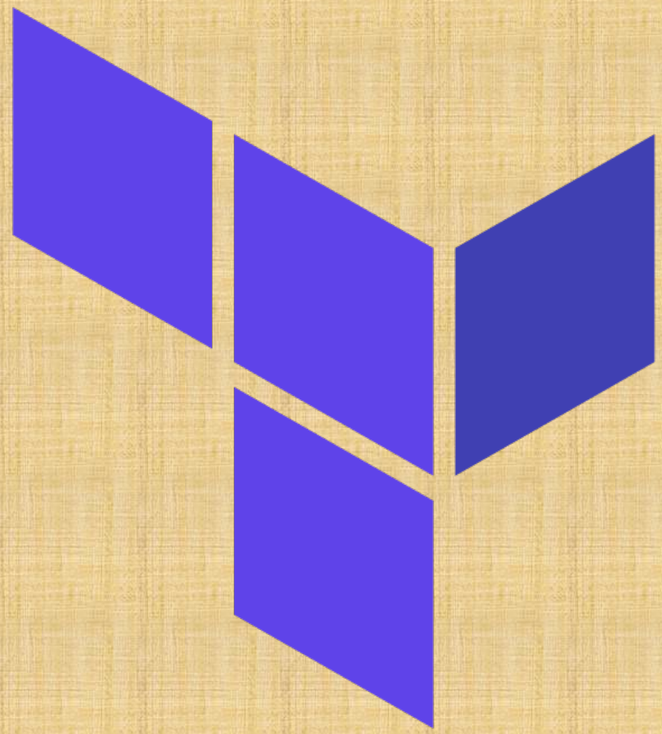
Terraform Cloud – Private Module Registry



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Cloud
CLI Driven Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

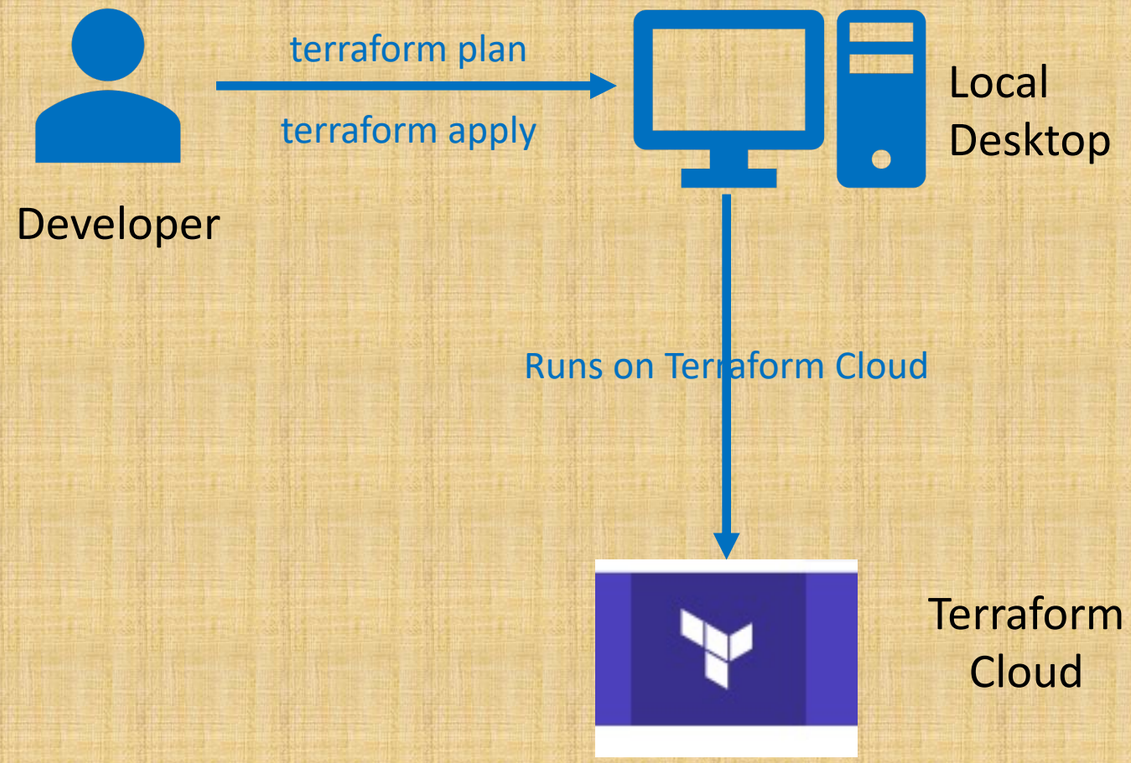
Terraform Cloud - CLI Driven Workflow

The CLI-driven run workflow uses Terraform's standard CLI tools to execute runs in Terraform Cloud.

The Terraform remote backend brings Terraform Cloud's collaboration features into the familiar Terraform CLI workflow

Users can start runs with the standard terraform plan and terraform apply commands, and can watch the progress of the run without leaving their terminal.

These runs execute remotely in Terraform Cloud; they use variables from the appropriate workspace, enforce any applicable Sentinel policies, and can access Terraform Cloud's private module registry and remote state inputs.



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud - CLI Driven Workflow

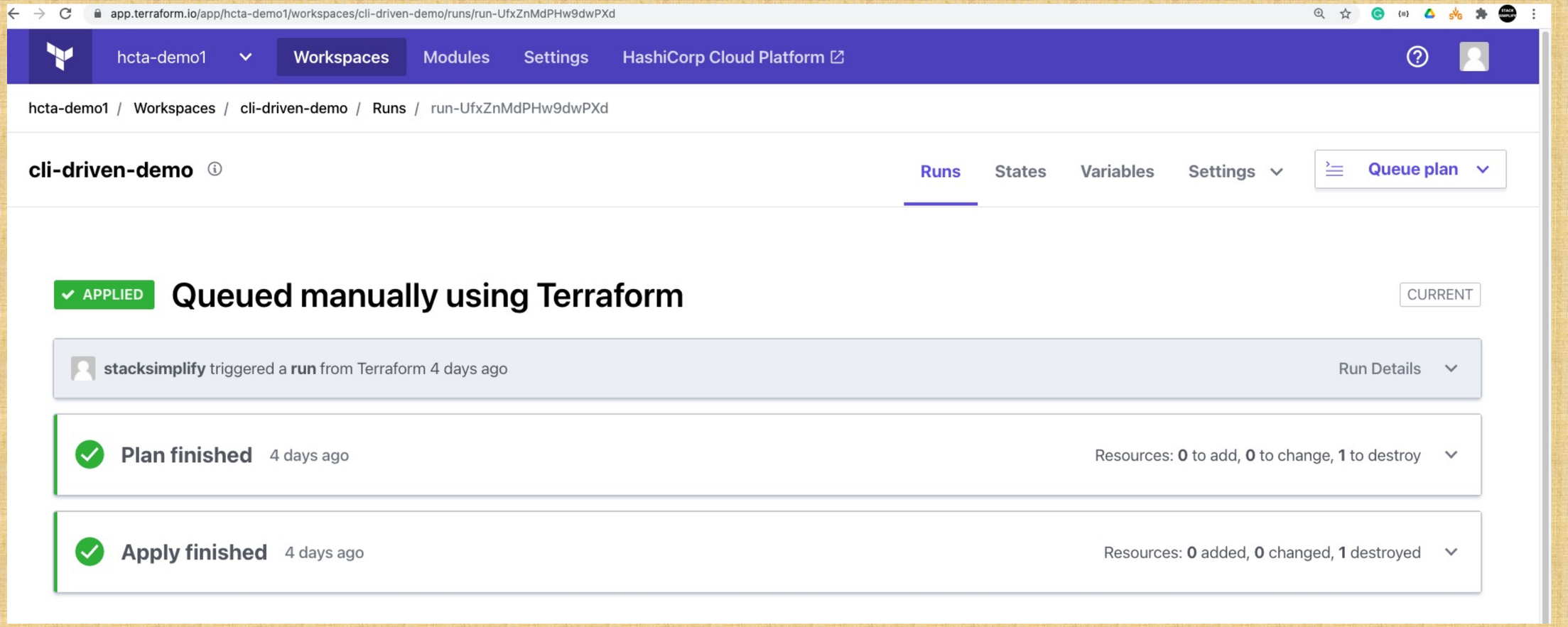
```
# Terraform Block
terraform {
  required_version = "~> 0.14" # which means any version equa
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
  # Update Terraform Cloud Backend Block Information below
  backend "remote" {
    organization = "hcta-demo1"

    workspaces {
      name = "cli-driven-demo"
    }
  }
}
```

Terraform Cloud as Backend configured in Terraform Block

Author: Nho Luong

Skill: DevOps Engineer Lead

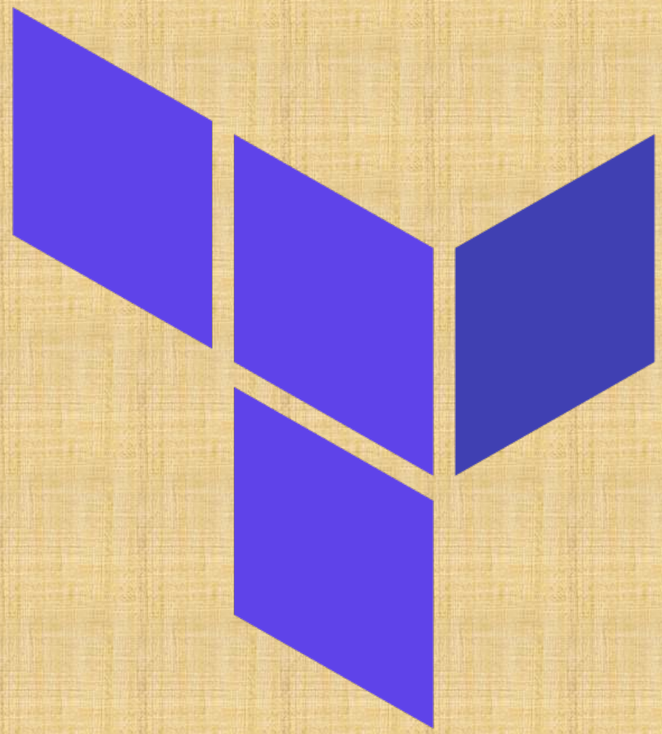
Terraform Cloud - CLI Driven Workflow



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Cloud Sentinel



Author: Nho Luong

Skill: DevOps Engineer Lead

What is Sentinel ?

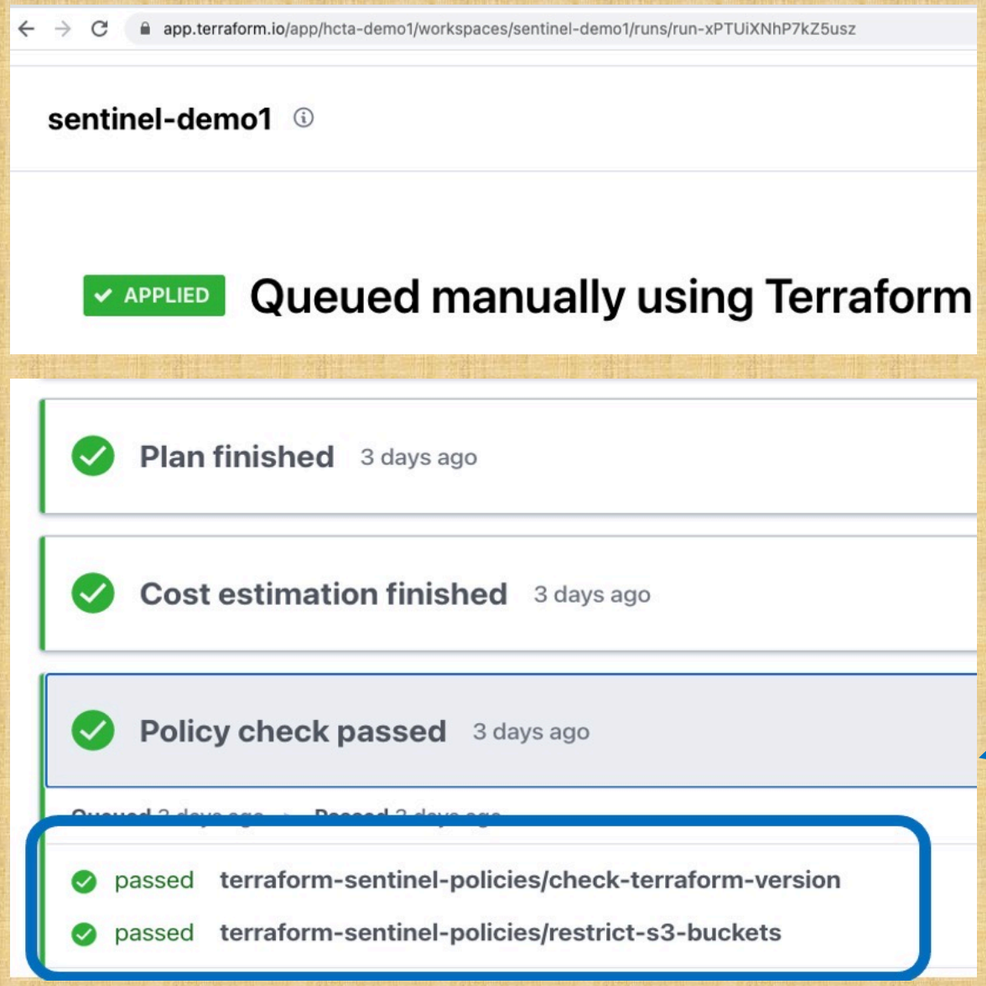
Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products.

It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

Author: Nho Luong

Skill: DevOps Engineer Lead

Sentinel – Basic Policies

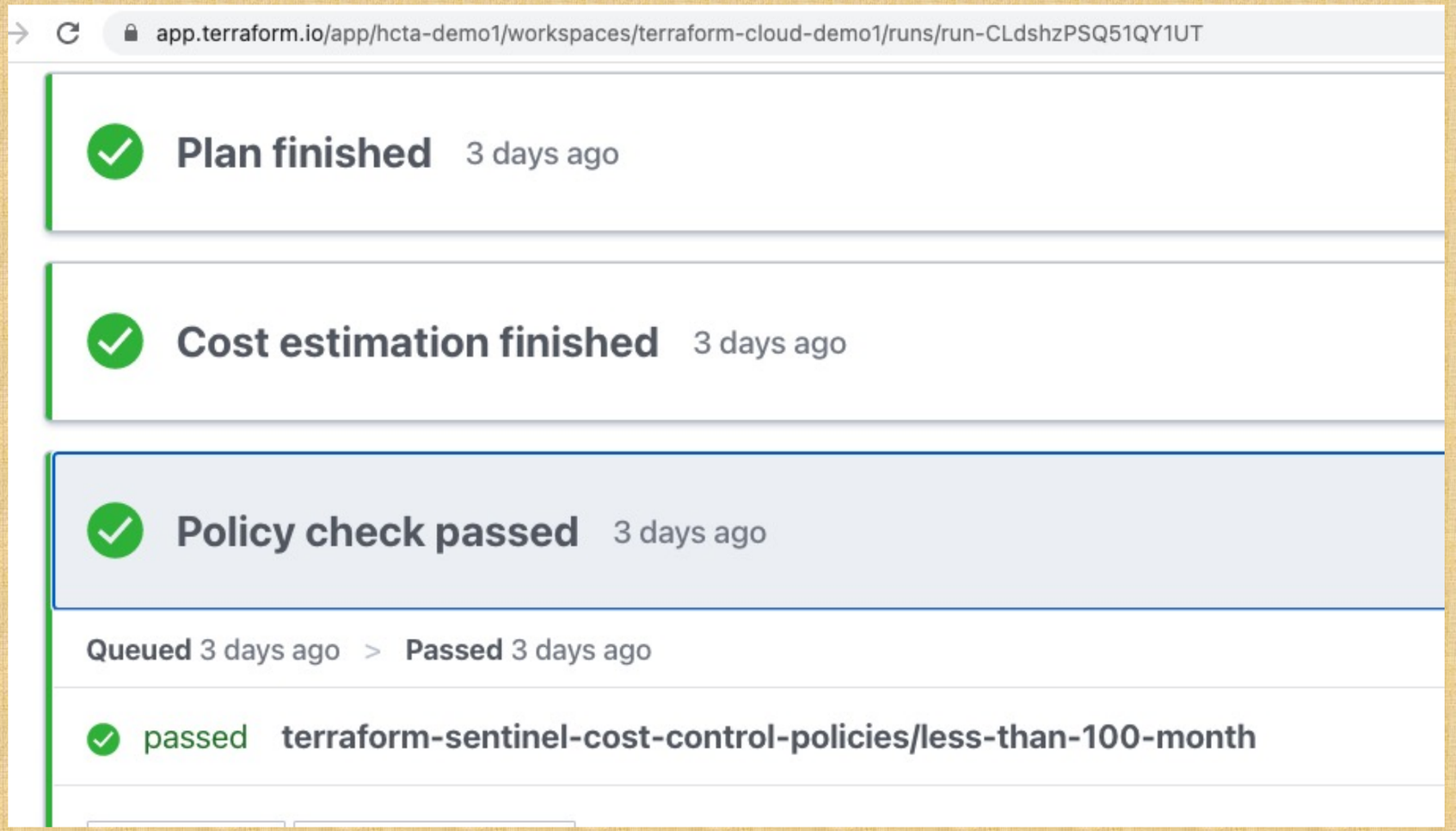


Sentinel Policies

Author: Nho Luong

Skill: DevOps Engineer Lead

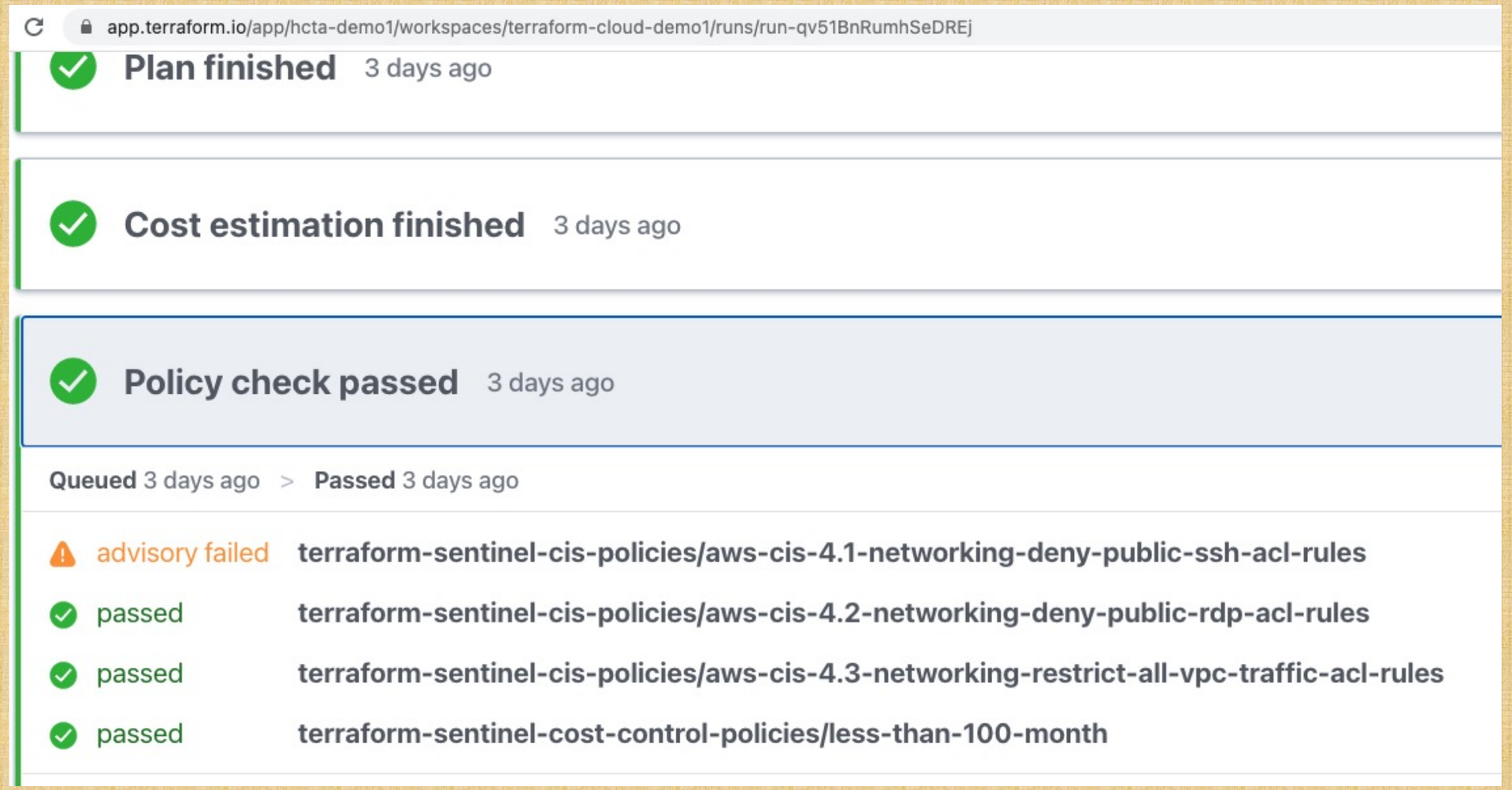
Sentinel – Cost Control Policies



Author: Nho Luong

Skill: DevOps Engineer Lead

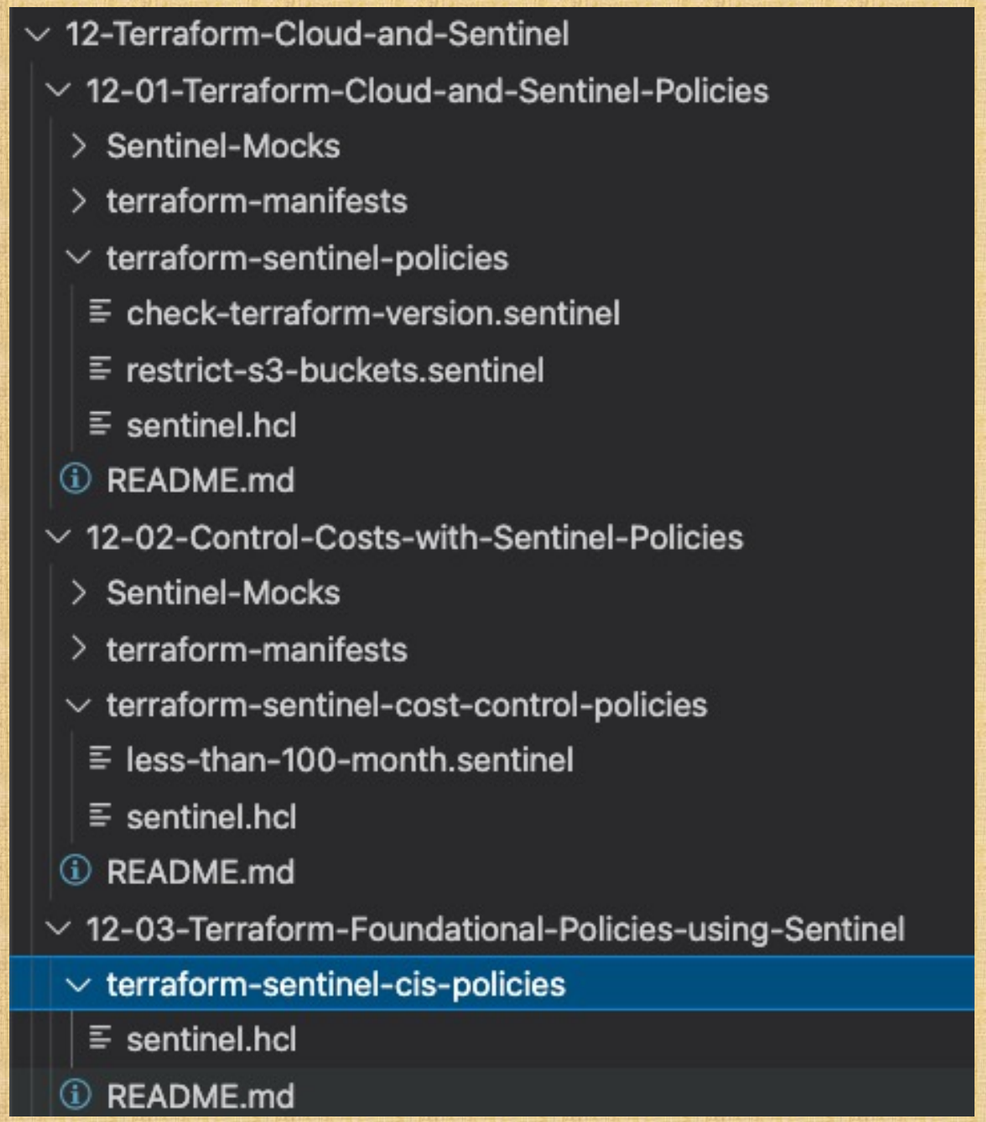
Sentinel – CIS Policies

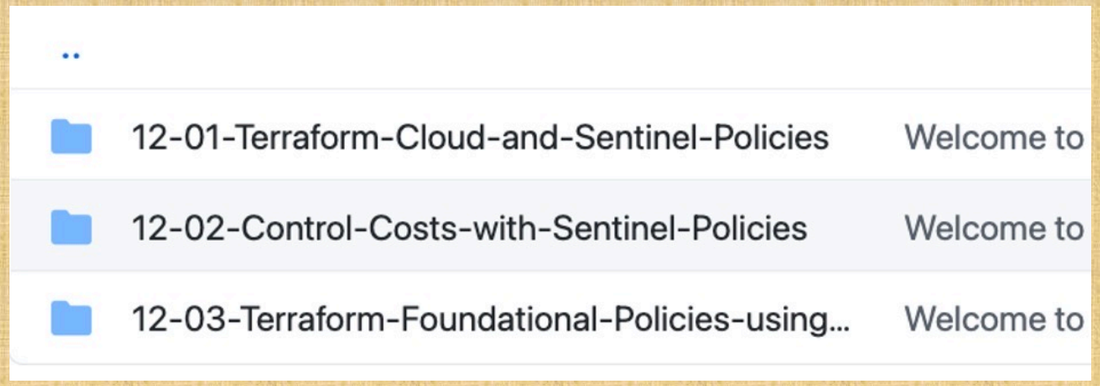


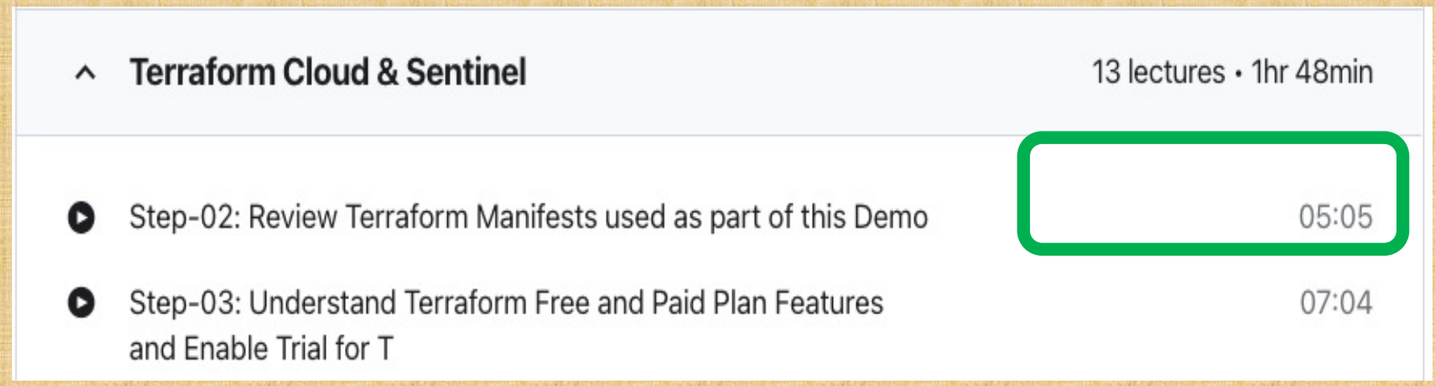
Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples with Step-by-Step Github Documentation



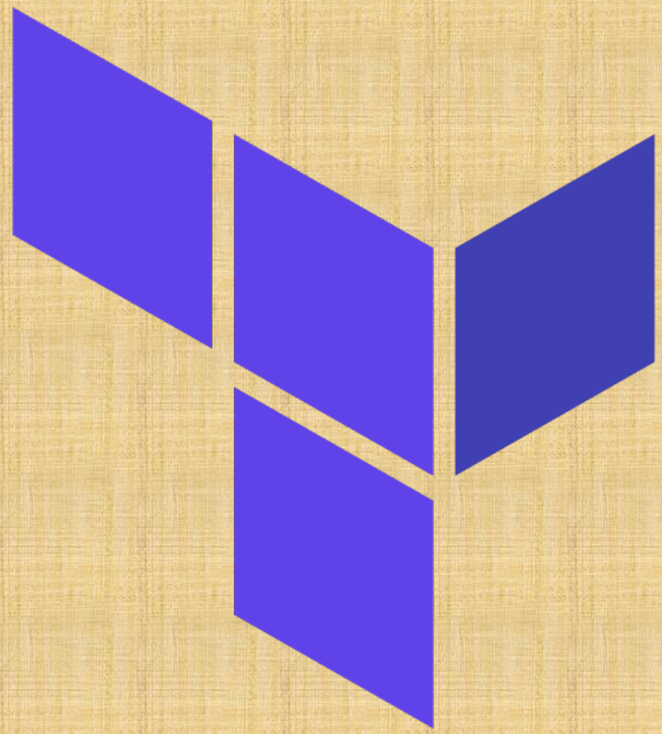




Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform
Expressions



Author: Nho Luong

Skill: DevOps Engineer Lead

Terraform Expressions

Expressions are used to refer to or compute values within a configuration

You can experiment with the behavior of Terraform's expressions from the Terraform expression console, by running the terraform console command.

Terraform Functions

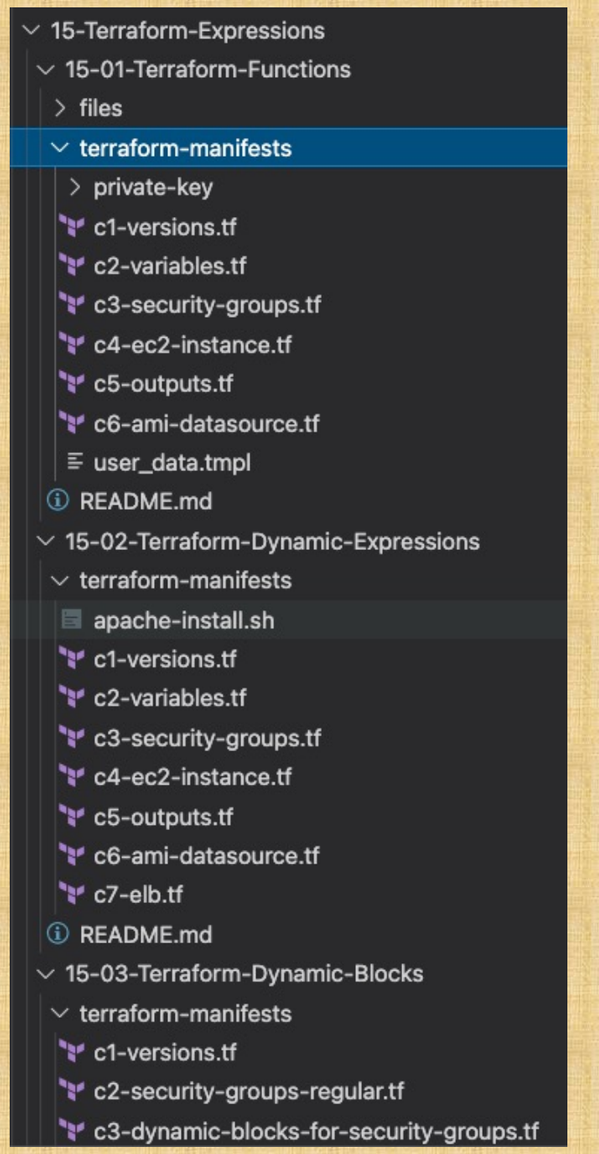
Terraform Dynamic Expressions

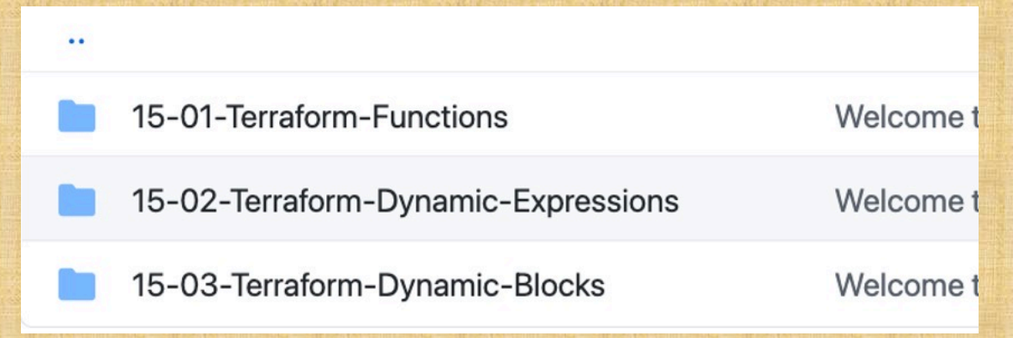
Terraform Dynamic Blocks

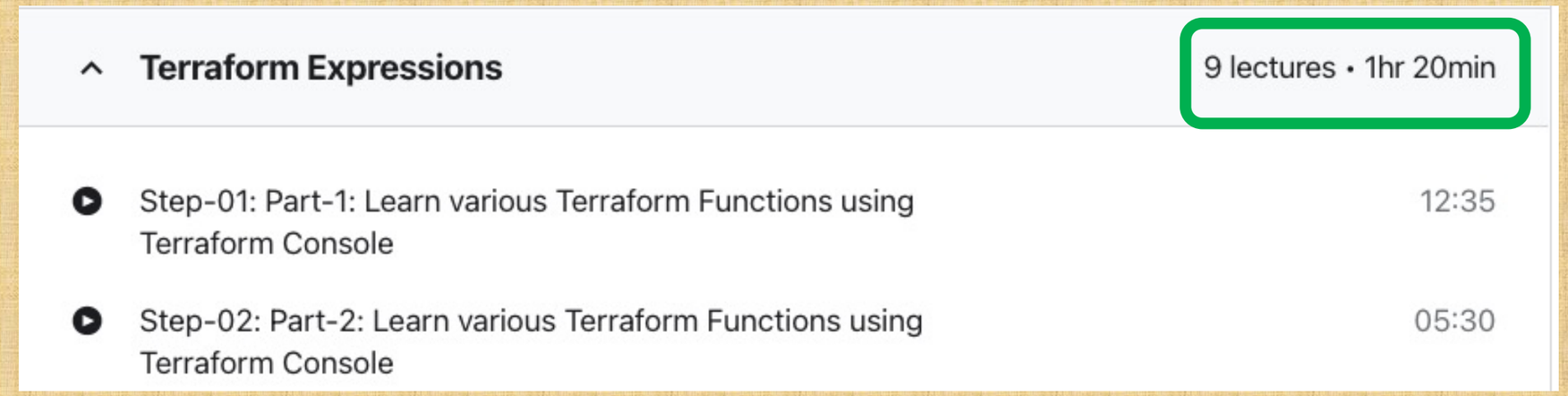
Author: Nho Luong

Skill: DevOps Engineer Lead

Practical Examples with Step-by-Step Github Documentation







Author: Nho Luong

Skill: DevOps Engineer Lead

ALL LIVE SLIDES ARE BEFORE THIS SLIDE

Author: Nho Luong

Skill: DevOps Engineer Lead

Thank You

Author: Nho Luong

Skill: DevOps Engineer Lead