

Terraform on AWS Associate

Author: Nho Luong

Skill: DevOps Engineer Lead



Screenshot of the AWS Certmetrics portal (cp.certmetrics.com/amazon/en/credentials/status) showing the "Certification status" page. The page displays four active certifications for the user "NHO LUONG" (AWS04440050):

- PROFESSIONAL**: **AWS Certified Solutions Architect - Professional**. SAP. Active since 2024-06-21, expires 2027-06-21. [View More](#)
- SPECIALTY**: **AWS Certified Security - Specialty**. SCS. Active since 2024-06-19, expires 2027-06-19. [View More](#)
- PROFESSIONAL**: **AWS Certified DevOps Engineer - Professional**. DOP. Active since 2024-06-17, expires 2027-06-17. [View More](#)
- ASSOCIATE**: **AWS Certified Solutions Architect - Associate**. SAA. Active since 2024-06-15, expires 2027-06-21. [View More](#)

The sidebar on the left includes links for HOME, PROFILE, EXAM REGISTRATION, EXAM HISTORY, CERTIFICATIONS (selected), BENEFITS, DIGITAL BADGES, and SUPPORT AND FAQS.



AWS EKS
Kubernetes

Azure AKS
Kubernetes

AWS ECS
Docker on AWS

AWS
CloudFormation

Google GKE
Kubernetes

AWS
Lambda & Serverless

TECHSTACK

**DevOps &
SRE
Roadmap**

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**

Terraform

40 Demo's

Basic Workflow

Providers

Resources

Variables

Datasources

State

Workspaces

Provisioners

Modules

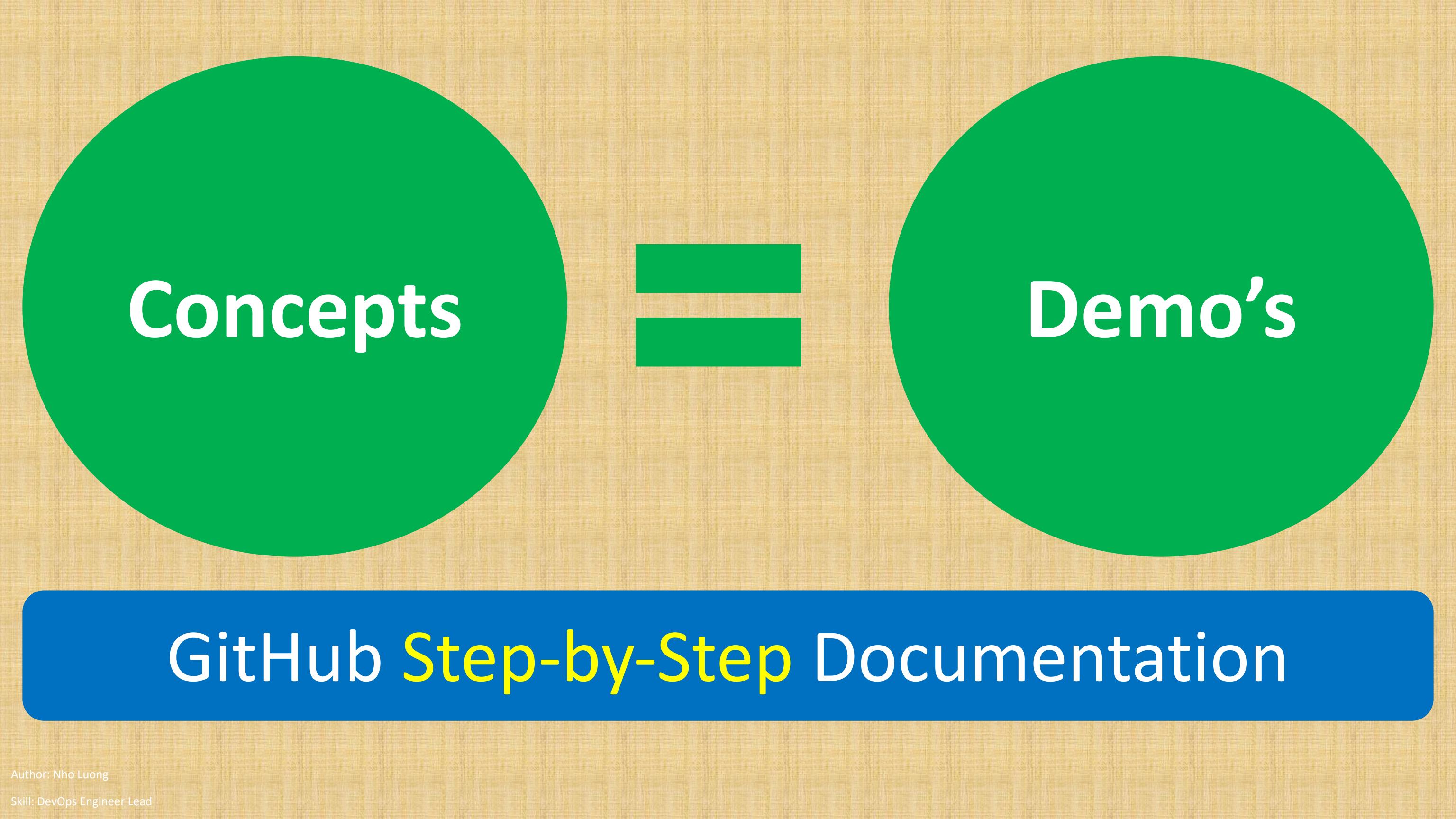
Cloud

Sentinel

State Import

Graph

Expressions



Concepts

Demo's

GitHub Step-by-Step Documentation

✓ hashicorp-certified-terraform-associate
> 01-Infrastructure-as-Code-IaC-Basics
> 02-Terraform-Basics
> 03-Terraform-Fundamental-Blocks
✓ 04-Terraform-Resources
✓ 04-01-Resource-Syntax-and-Behavior
✓ terraform-manifests
└ c1-versions.tf
└ c2-ec2-instance.tf
└ README.md
✓ 04-02-Meta-Argument-depends_on
✓ terraform-manifests
└ apache-install.sh
└ c1-versions.tf
└ c2-vpc.tf
└ c3-ec2-instance.tf
└ c4-elastic-ip.tf
└ README.md
> 04-03-Meta-Argument-count
> 04-04-Meta-Argument-for_each
> 04-05-Meta-Argument-lifecycle
> 04-06-Provisioners
> 05-Terraform-Variables
> 06-Terraform-Datasources
> 07-Terraform-State
> 08-Terraform-Workspaces
> 09-Terraform-Provisioners
> 10-Terraform-Modules
> 11-Terraform-Cloud-and-Enterprise-Capabilities
> 12-Terraform-Cloud-and-Sentinel

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

- 01-Infrastructure-as-Code-ia...
- 02-Terraform-Basics
- 03-Terraform-Fundamental-B...
- 04-Terraform-Resources
- 05-Terraform-Variables
- 06-Terraform-Datasources
- 07-Terraform-State
- 08-Terraform-Workspaces

- 09-Terraform-Provisioners
- 10-Terraform-Modules
- 11-Terraform-Cloud-and-Ente...
- 12-Terraform-Cloud-and-Sen...
- 13-Terraform-State-Import
- 14-Terraform-Graph
- 15-Terraform-Expressions

Terraform Providers Used



AWS EC2 Instances

AWS VPC

AWS Security Groups

AWS S3 Buckets

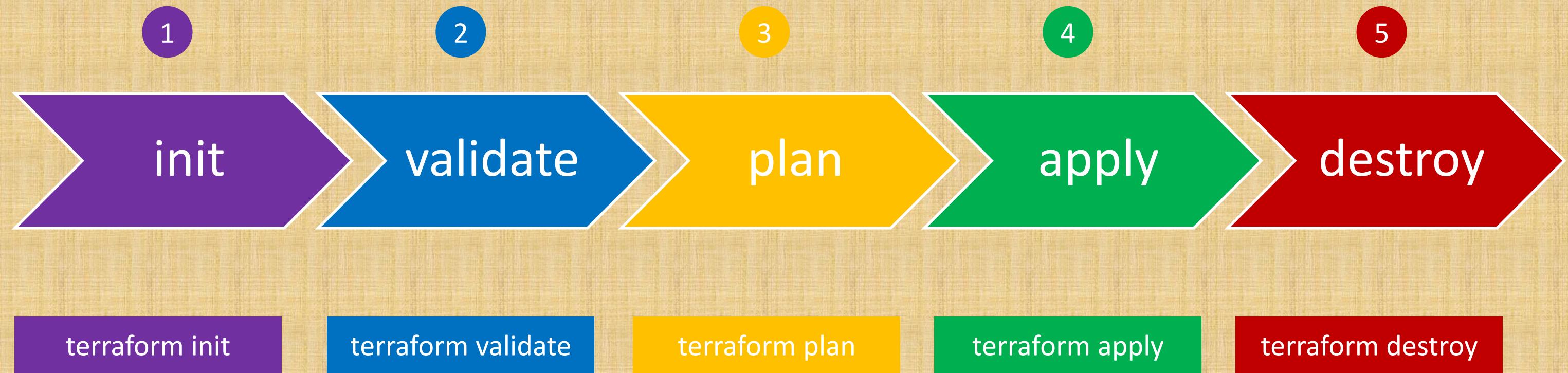
AWS IAM Users

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

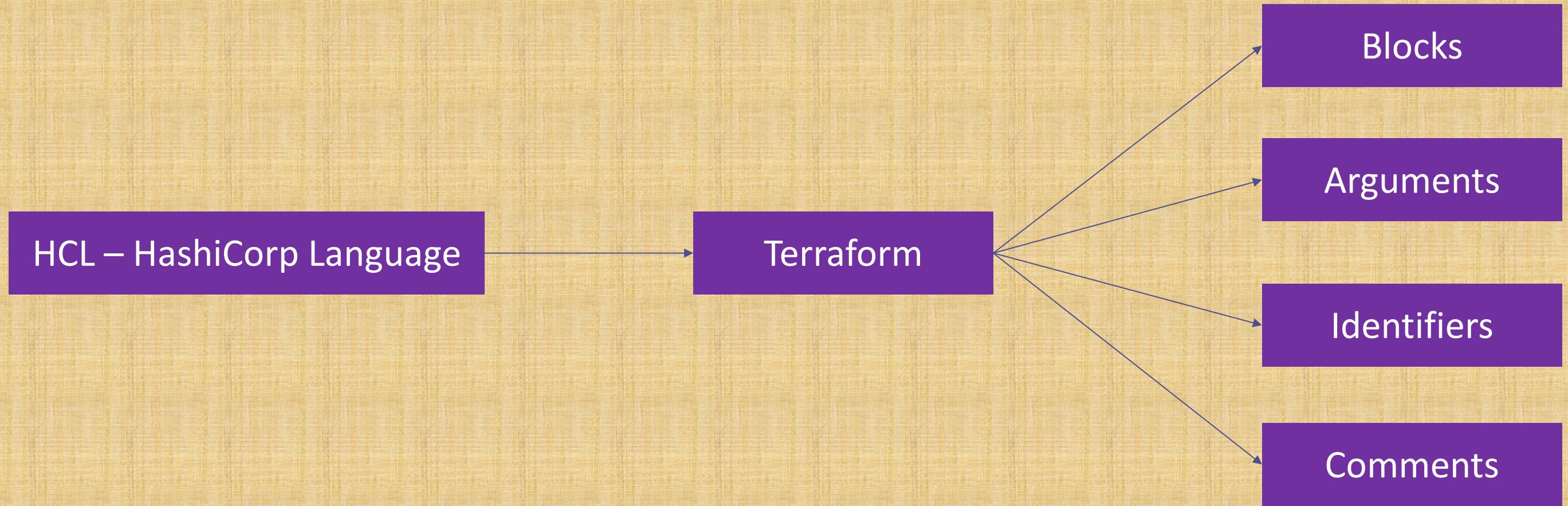
GitHub Repositories

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

Terraform Workflow



Terraform Language Basics – Configuration Syntax



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.

Terraform Block

Providers Block

Resources Block

Fundamental Blocks

Input Variables Block

Output Values Block

Local Values Block

Variable Blocks

Data Sources Block

Modules Block

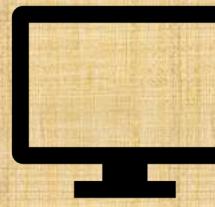
Calling / Referencing Blocks

Terraform Providers

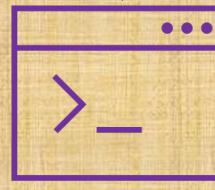
Terraform Admin



Local Desktop



Terraform CLI



Terraform AWS Provider

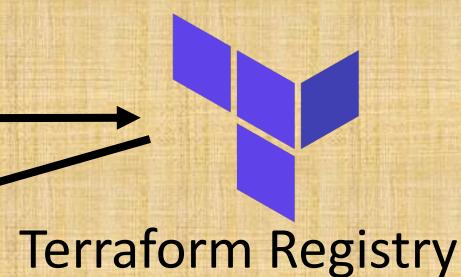


2 terraform validate

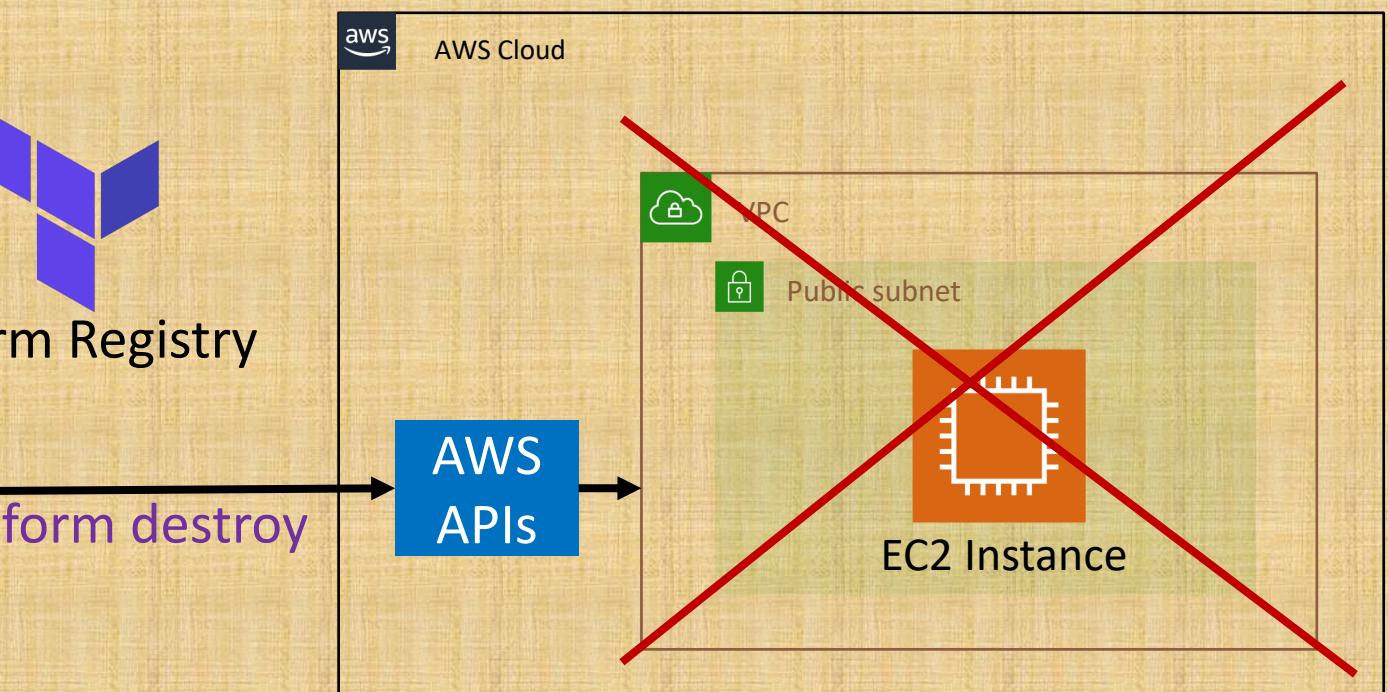
3 terraform plan

1 terraform init

Download Provider



4 terraform apply
5 terraform destroy



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

Terraform Resources

Terraform
Language Basics

Terraform
Resource Syntax

Terraform
Resource Behavior

Terraform
State

Resource
Meta-Argument
count

Resource
Meta-Argument
depends_on

Resource
Meta-Argument
for_each

Resource
Meta-Argument
lifecycle

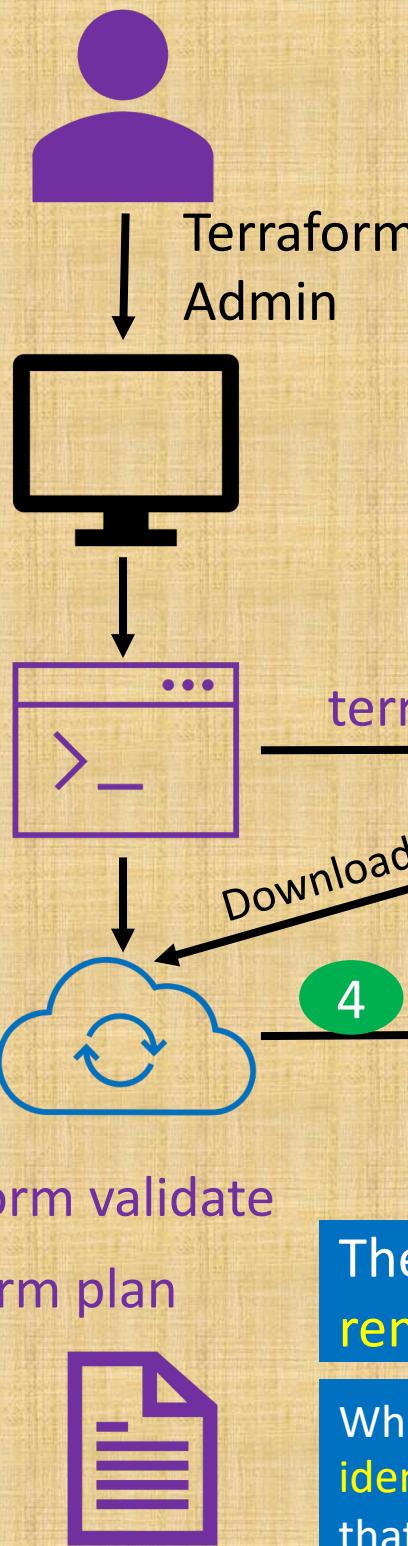
Terraform State

Local Desktop

Terraform CLI

Terraform AWS Provider

Terraform State File
`terraform.tfstate`



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.

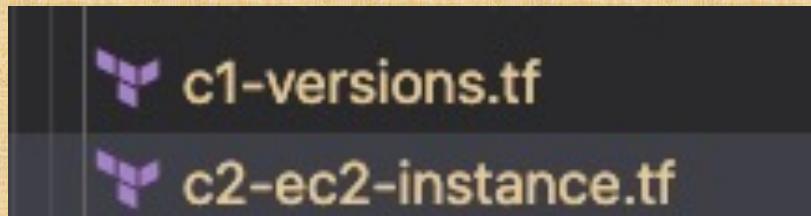
This state is stored by default in a local file named "**terraform.tfstate**", but it can also be stored **remotely**, which works better in a **team** environment.

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.

Desired & Current Terraform States

Terraform Configuration Files



Real World Resource – EC2 Instance

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
web	i-0663449fef49e9cf5	Running	t2.micro	2/2 checks ...	No alarms +	us-east-1b
Instance: i-0663449fef49e9cf5 (web)						
Details	Security	Networking	Storage	Status checks	Monitoring	Tags
Instance ID i-0663449fef49e9cf5 (web)	Public IPv4 address 54.144.73.100 open address	Private IPv4 addresses 172.31.94.137	Public IPv4 DNS ec2-54-144-73-100.compute-1.amazonaws.com open address	Private IPv4 DNS ip-172-31-94-137.ec2.internal	VPC ID vpc-54972d2e (default-vpc)	Subnet ID subnet-d2e590fc
Instance state Running	Elastic IP addresses -	AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations.	IAM Role -			

Desired State



Current State

Terraform Variables

Terraform
Input
Variables

Terraform
Output
Values

Terraform
Local
Values

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.

Input Variables - Basics

1

Provide Input Variables when prompted during **terraform plan** or **apply**

2

Override default variable values using CLI argument **-var**

3

Override default variable values using Environment Variables (**TF_var_aa**)

4

Provide Input Variables using **terraform.tfvars** files

5

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

Terraform
Input
Variables

Terraform State

Terraform
Remote
State
Storage

Terraform
Commands
from
State
Perspective

What is Terraform Backend ?

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

Terraform
State Storage



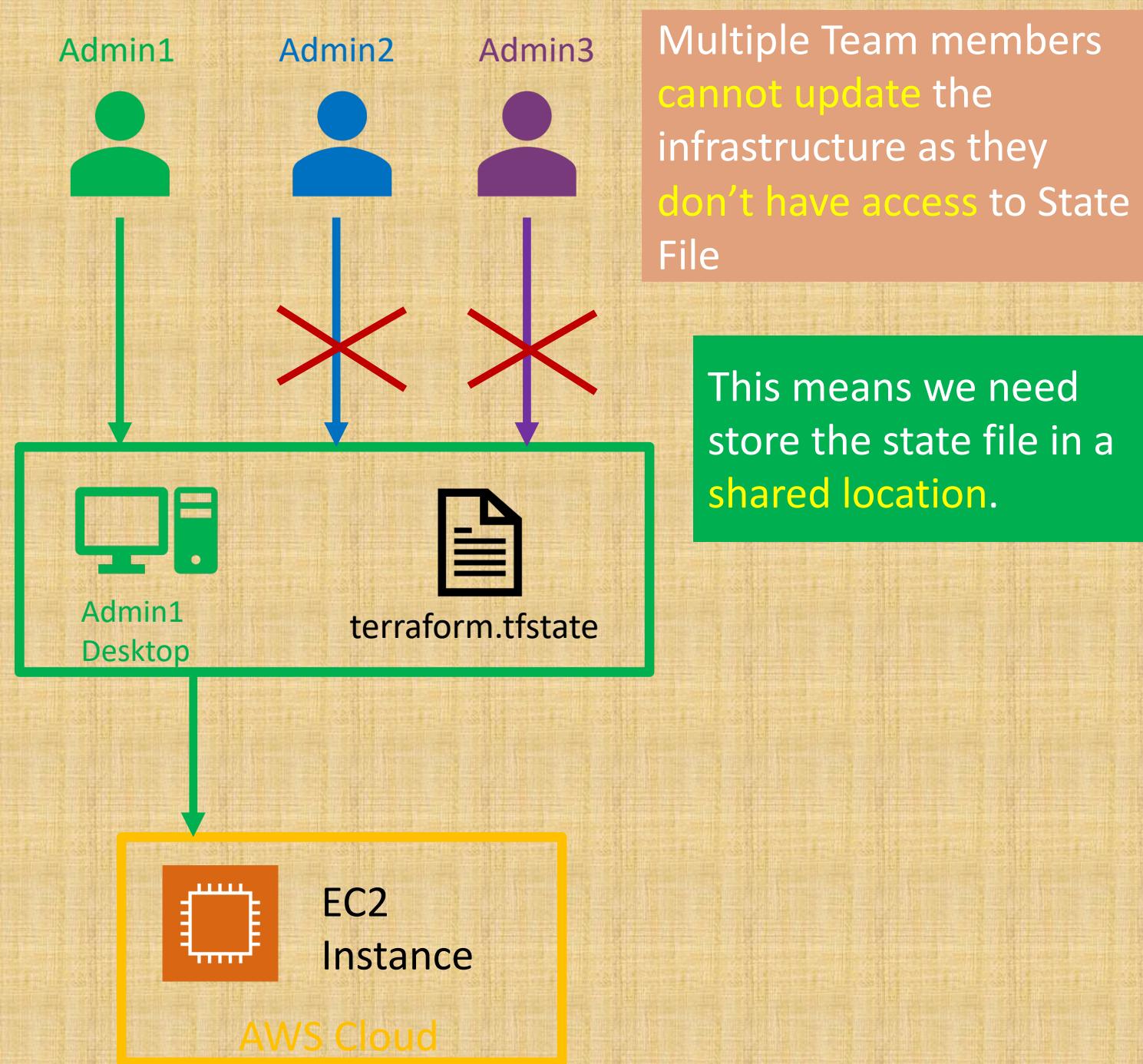
AWS S3 Bucket

Terraform
State Locking

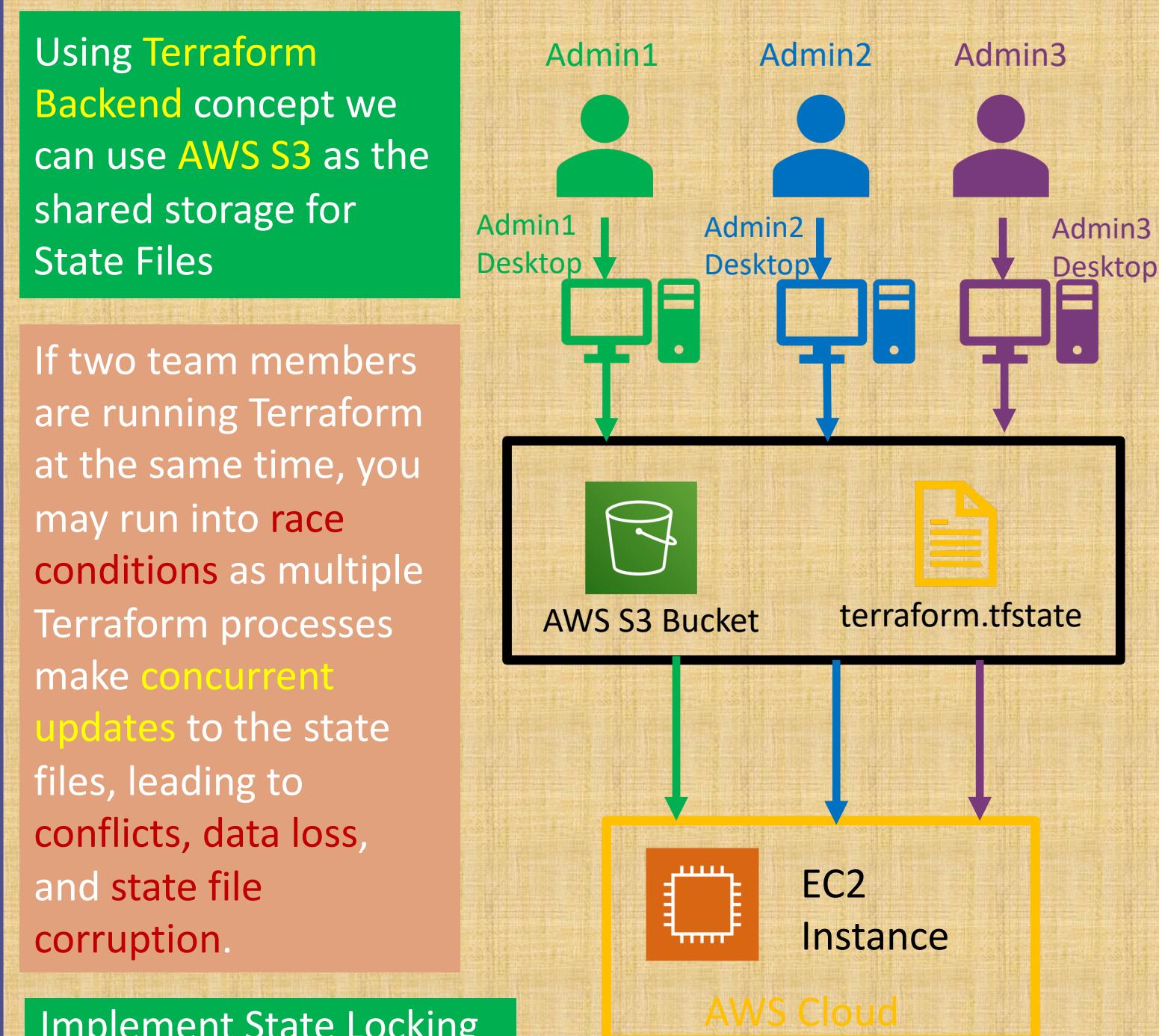


AWS DynamoDB

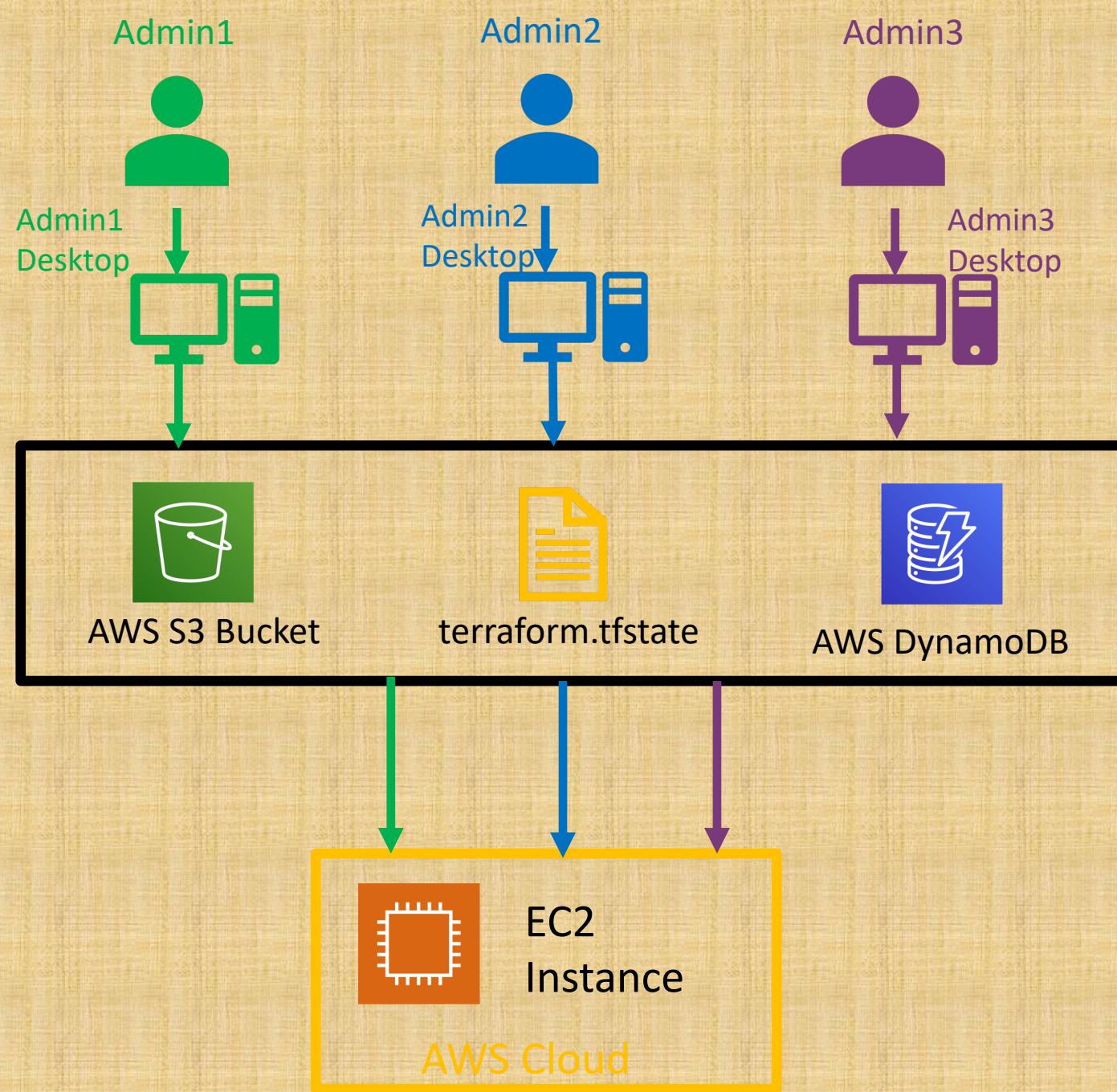
Local State File



Remote State File



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.

Terraform Commands – State Perspective

terraform show

terraform refresh

terraform plan

terraform state

Terraform
Commands

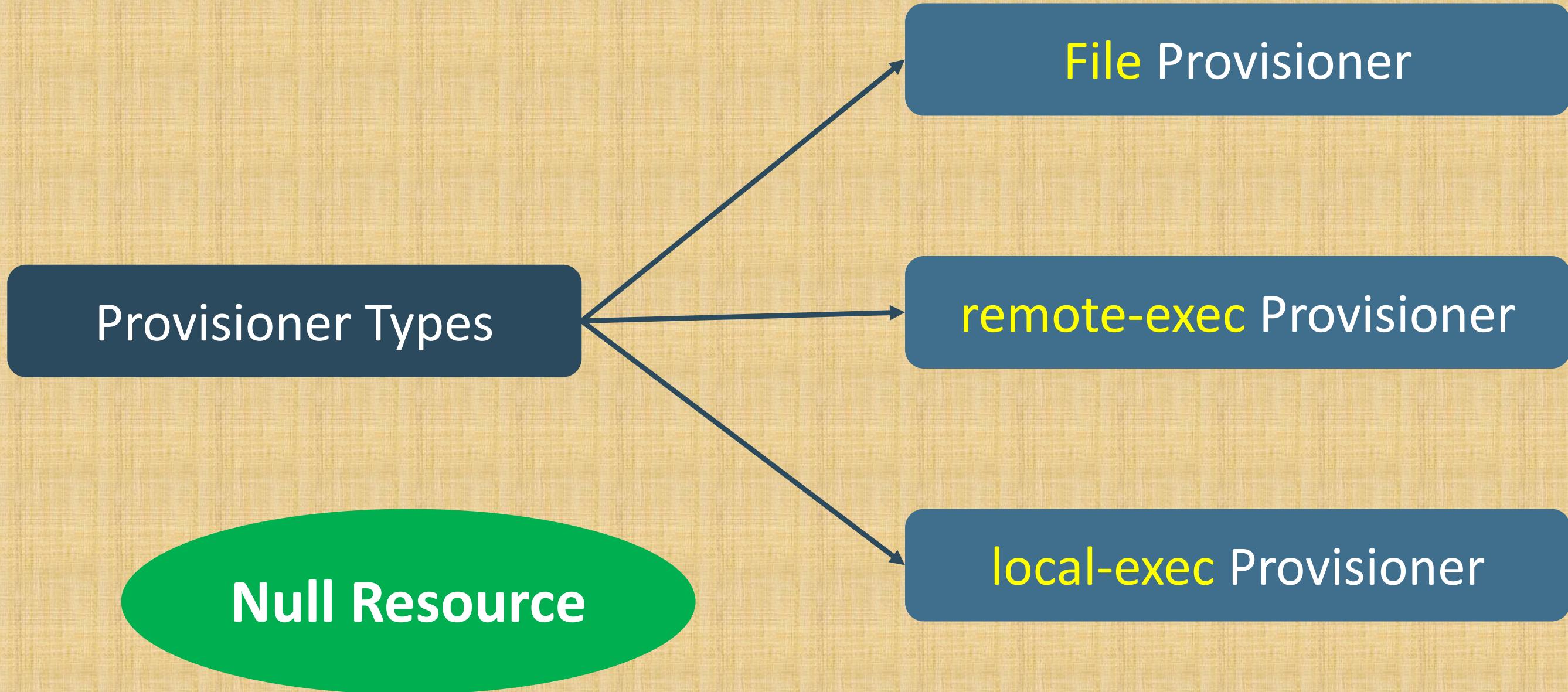
terraform
force-unlock

terraform taint

terraform untaint

terraform
apply target

Types of Provisioners

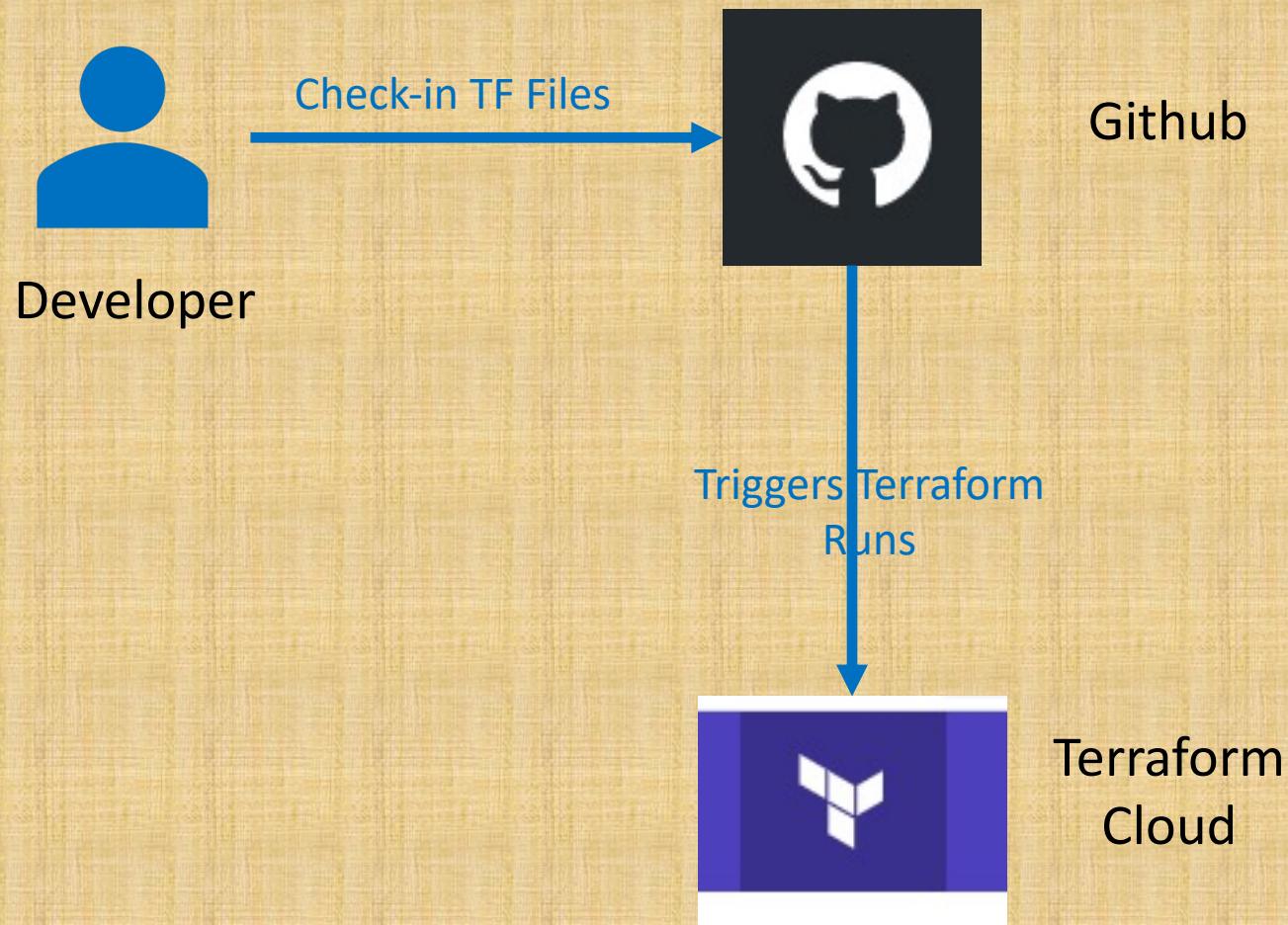


Terraform Modules

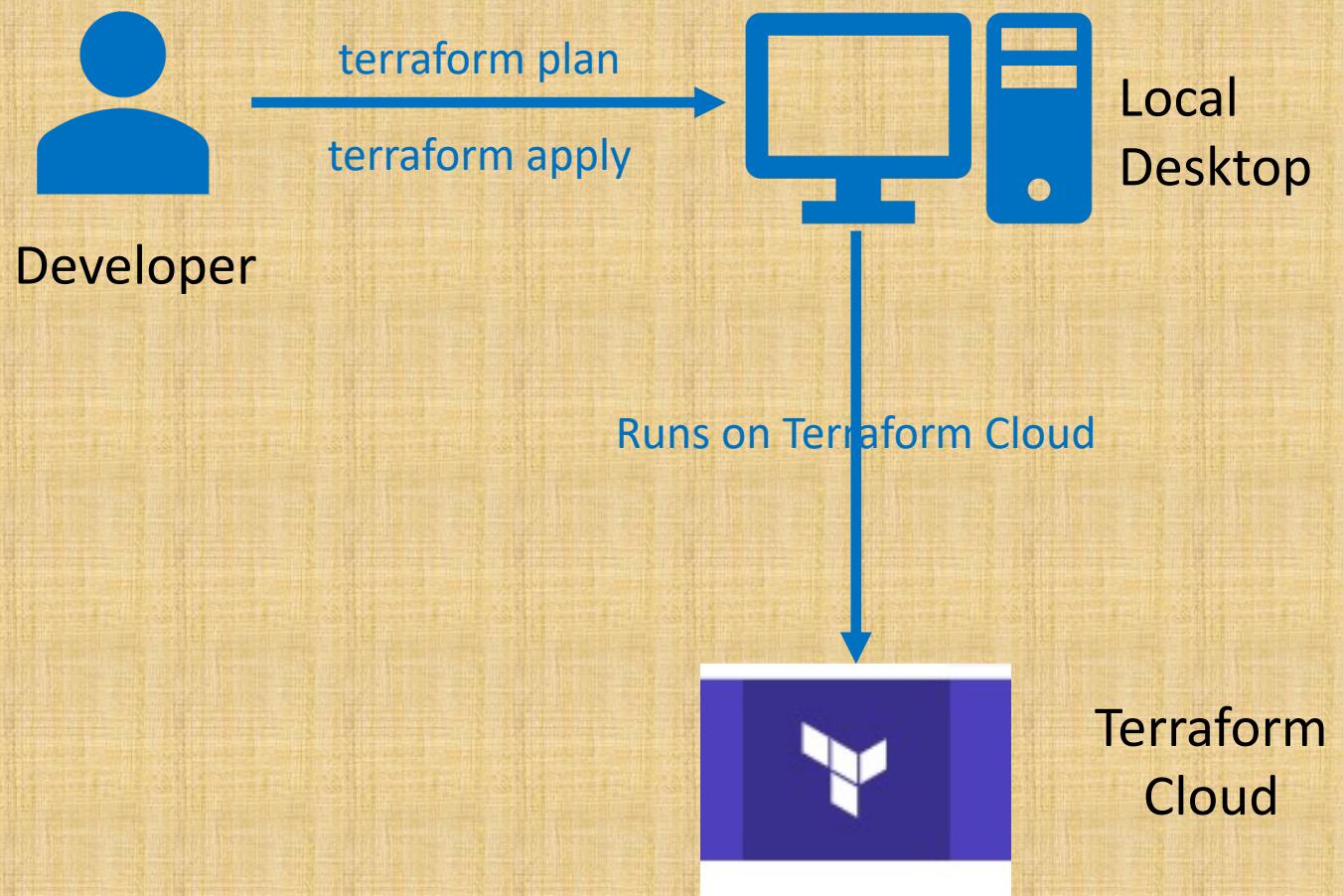
Implement by
calling a **Public
Module** from
Terraform
Registry

Build a
Terraform
Local Module

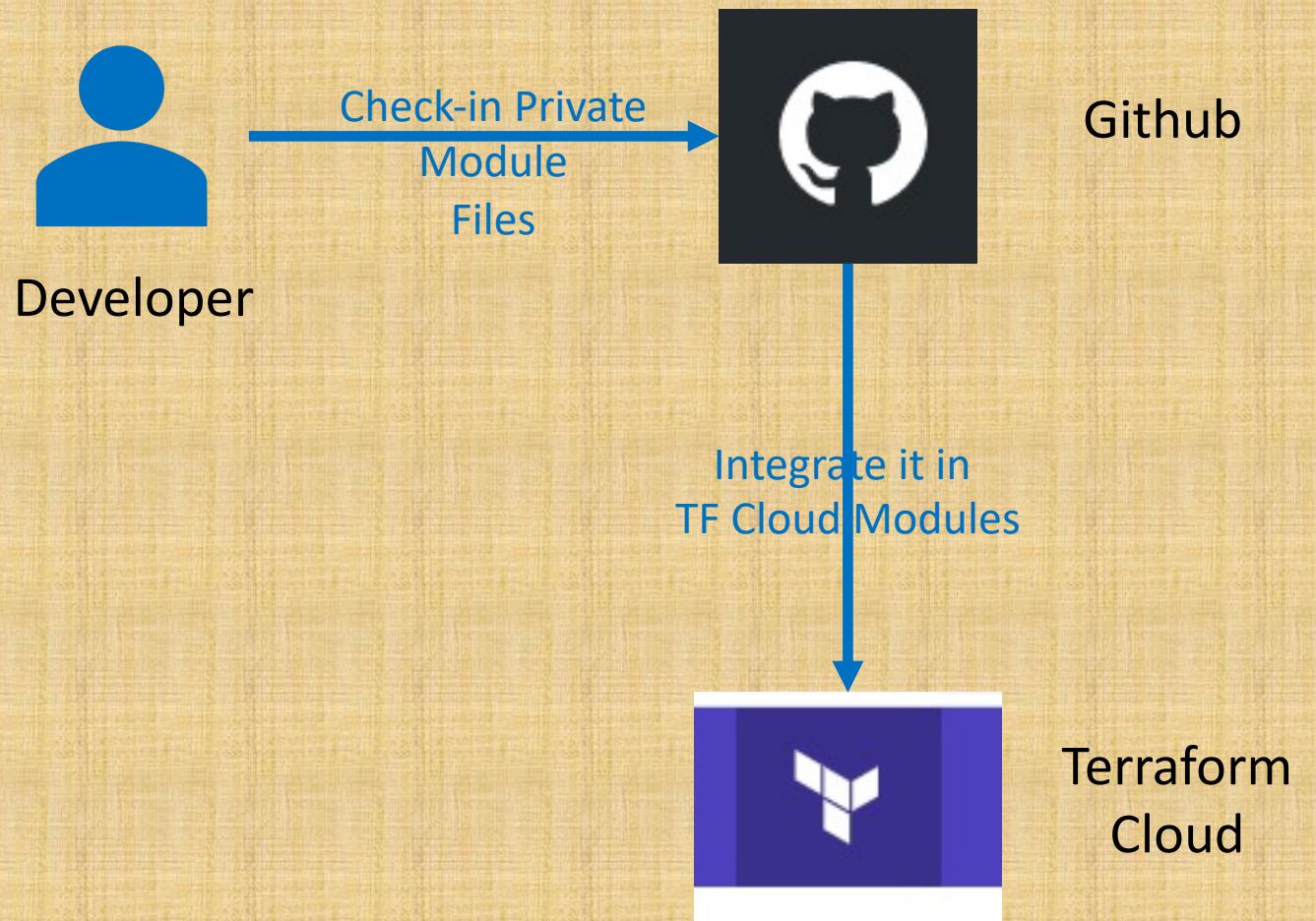
Terraform VCS-Driven Workflow



Terraform CLI-Driven Workflow



Publish Private Module Registry in Terraform Cloud



Terraform Cloud & Sentinel

Basic
Policies

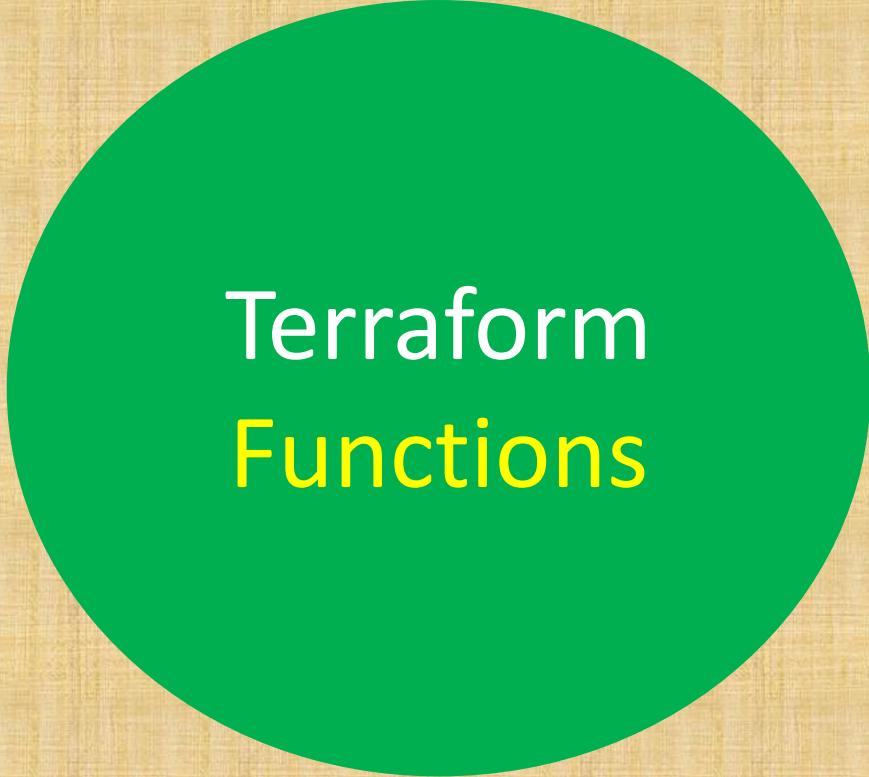
Cost Control
Policies

CIS
Policies

Terraform Expressions

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

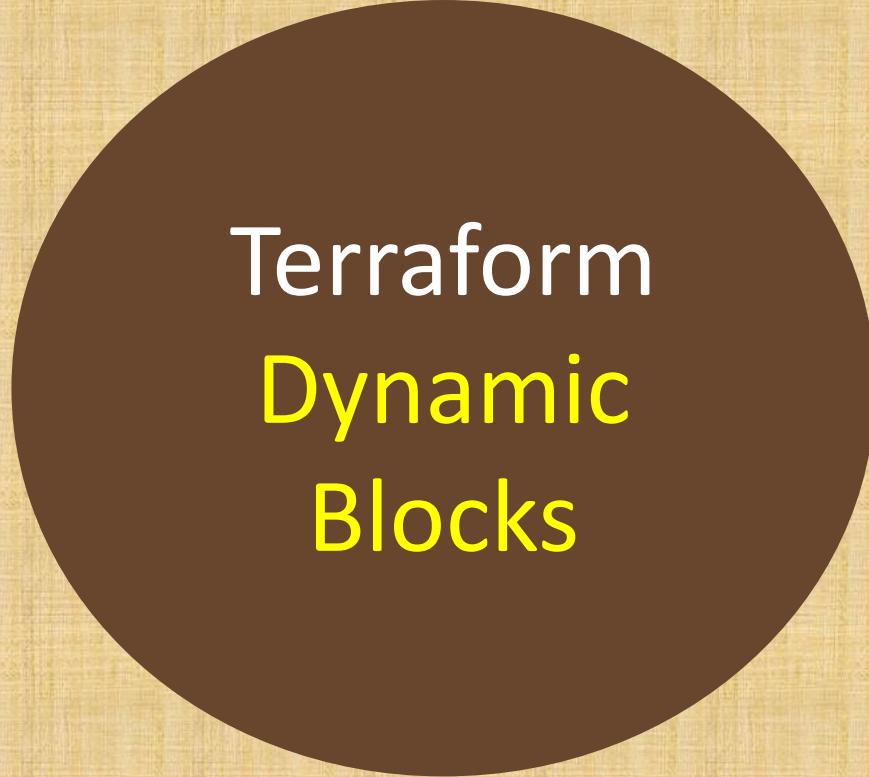
You can experiment with the behavior of Terraform's expressions from the Terraform expression console, by running the terraform console command.



Terraform
Functions



Terraform
Dynamic
Expressions

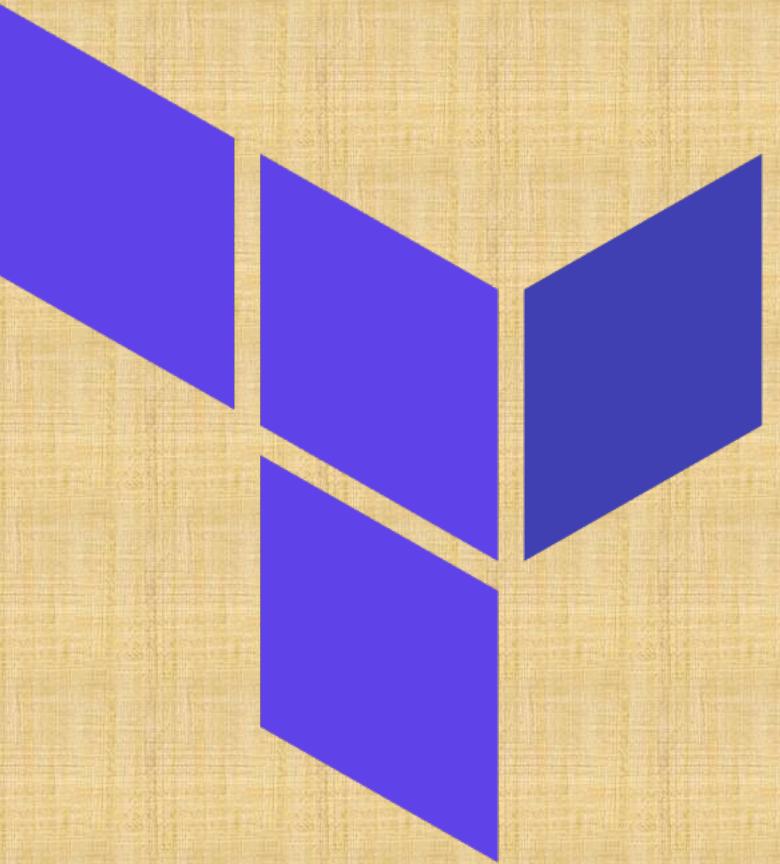


Terraform
Dynamic
Blocks

Terraform

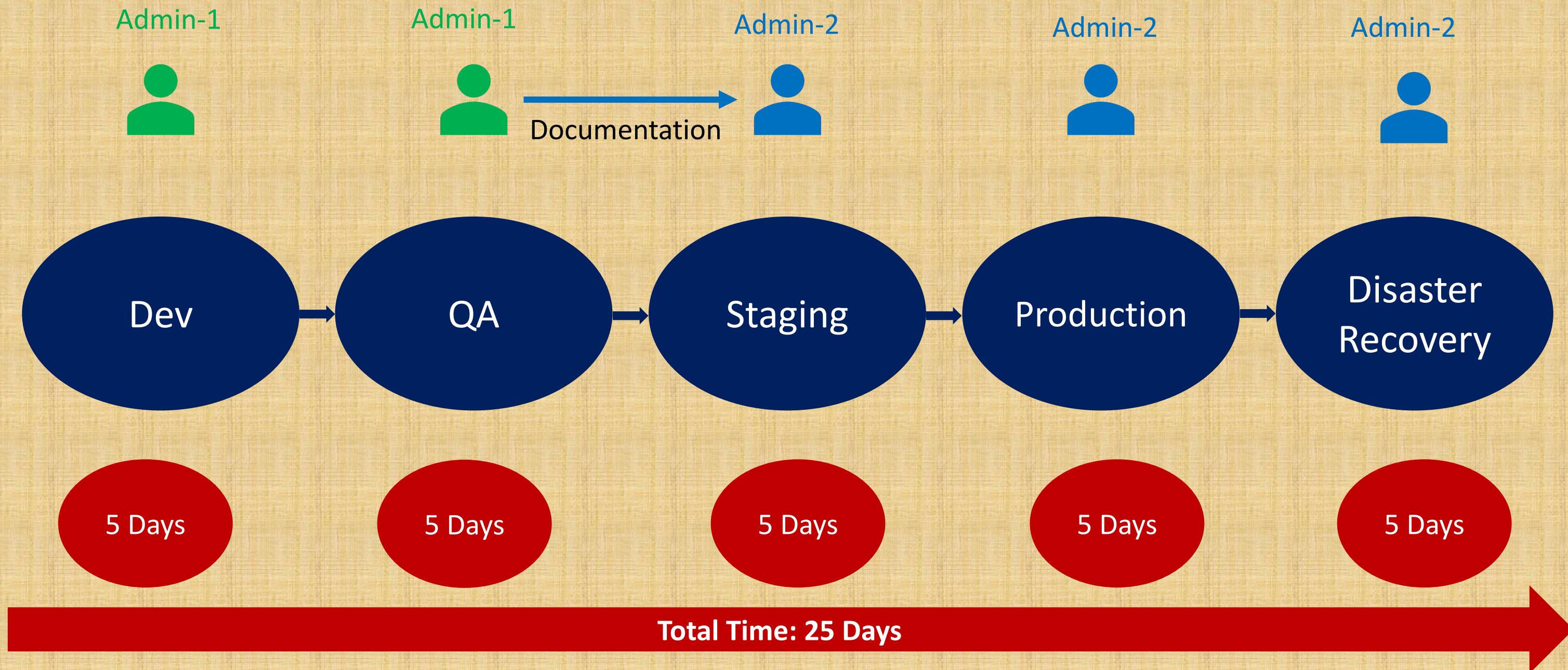
Infrastructure as Code

IaC



What is Infrastructure as Code ?

Traditional Way of Managing Infrastructure



Traditional Way of Managing Infrastructure

Admin-1



Admin-1



Admin-2



Admin-2



Admin-2



No CI

Delays

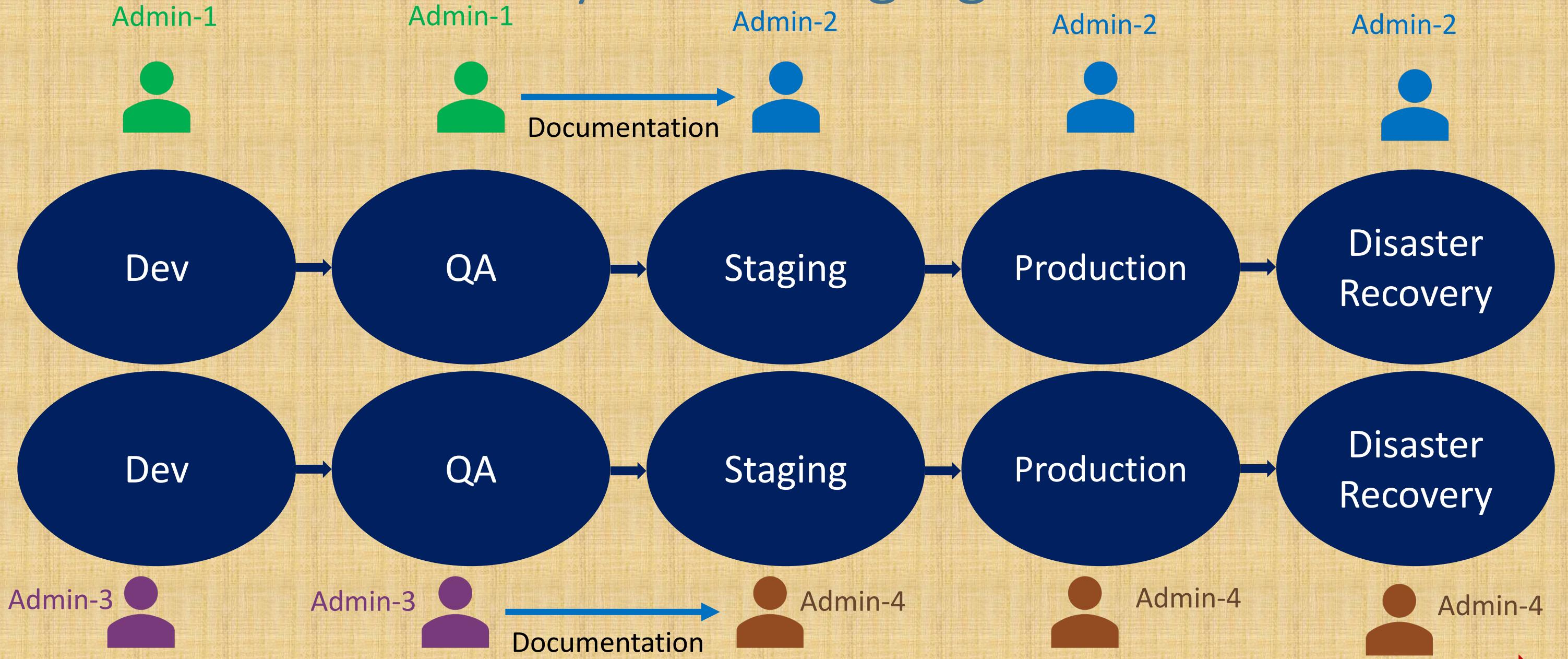
Issues

Outages

Not-in-Sync

Many Problems at many places in manual process

Traditional Way of Managing Infrastructure



Infrastructure scalability – Workforce need to be increased to meet the timelines

Traditional Way of Managing Infrastructure

Prod-1

Prod-2

Prod-3

Prod-4

Scale Up

Prod-1

Prod-2

Scale Down

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

5 Days

Admin-1



Check-In TF Code



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
Template
s

Quick &
Fast

Reliable

Tracked
for Audit

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

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.

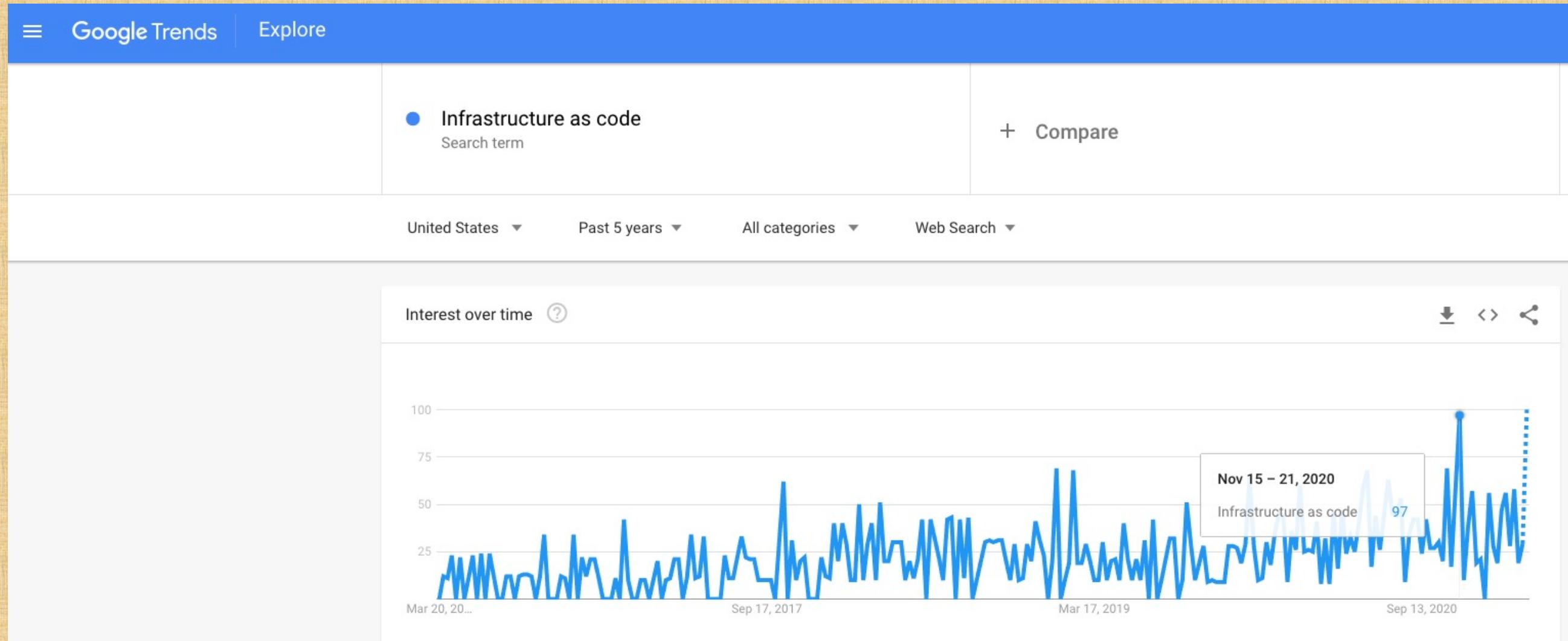
Security

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.

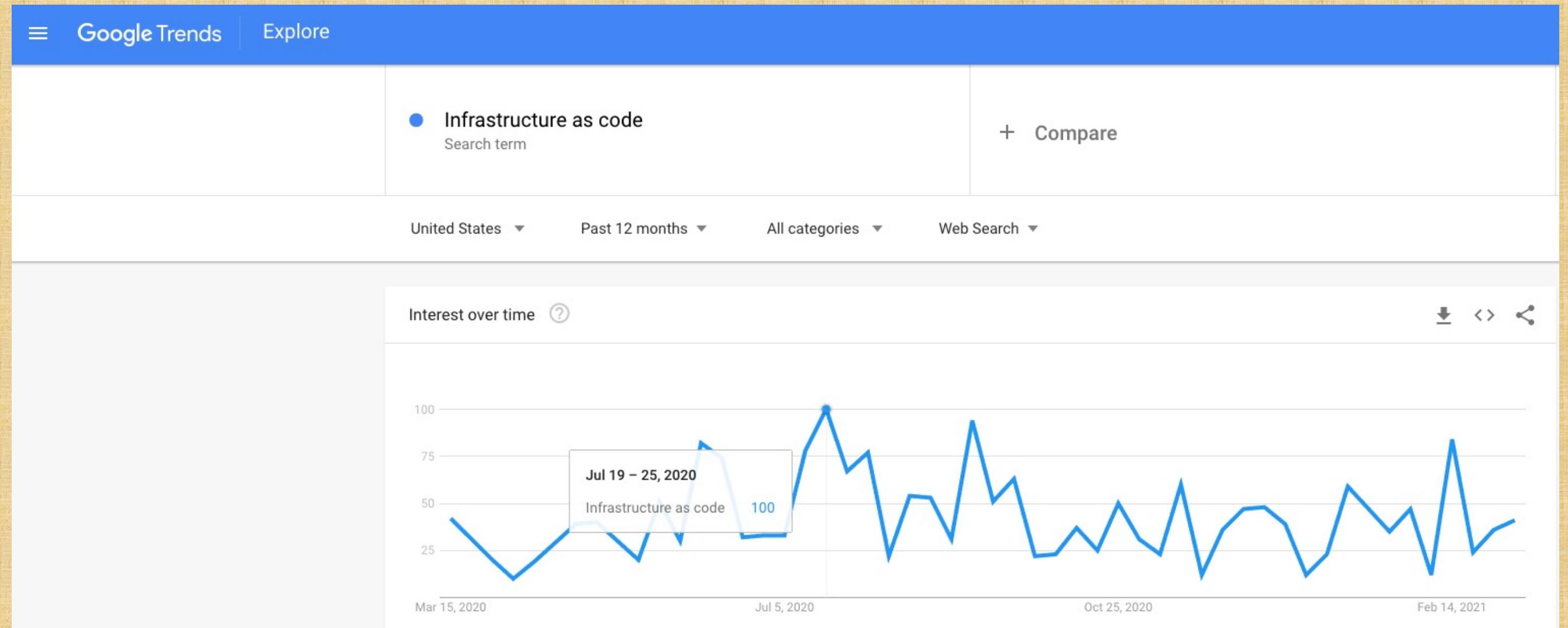
Audit

Terraform not only creates resources it also **maintains the record** of what is created in real world cloud environments using its State files.

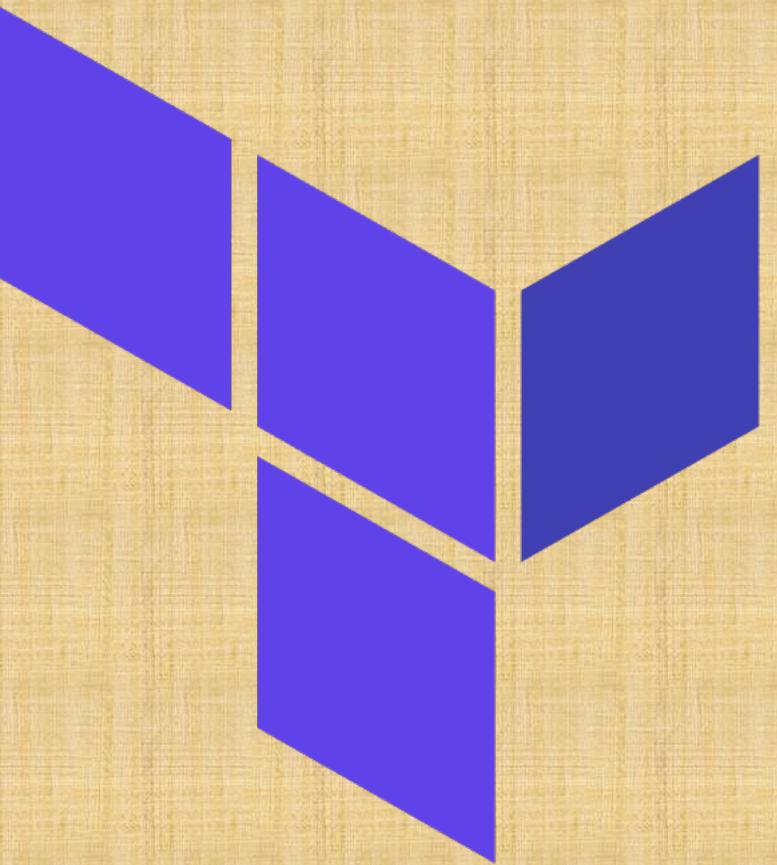
Google Trends – Past 5 Years



Google Trends – Past 1 Year



Terraform Installation



Terraform Installation

Terraform CLI

AWS CLI

VS Code Editor

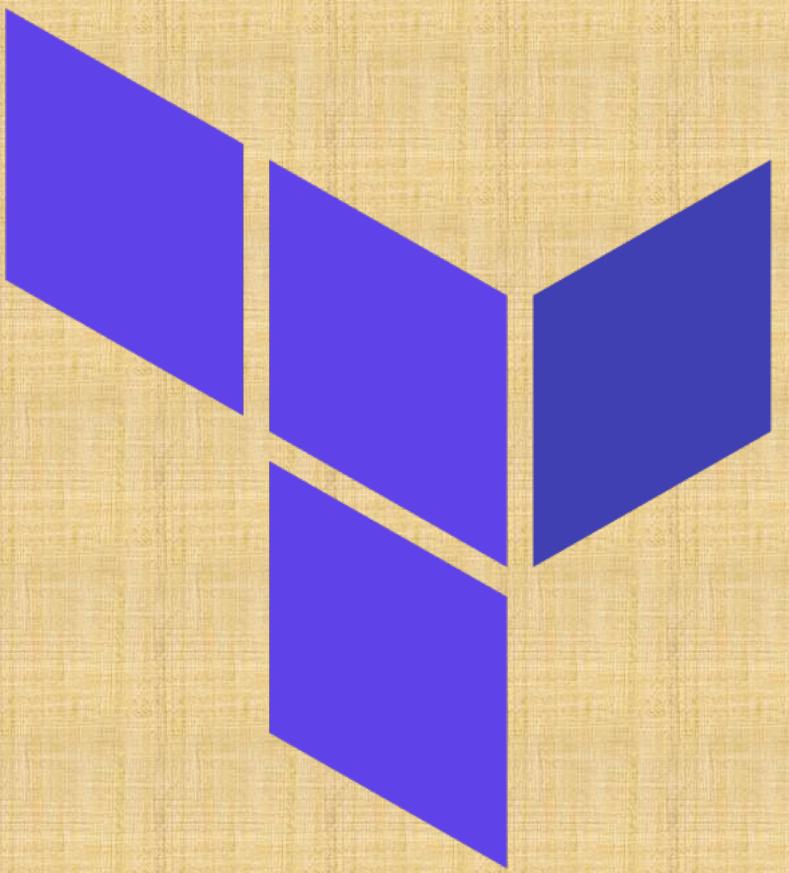
Terraform plugin
for VS Code

Mac OS

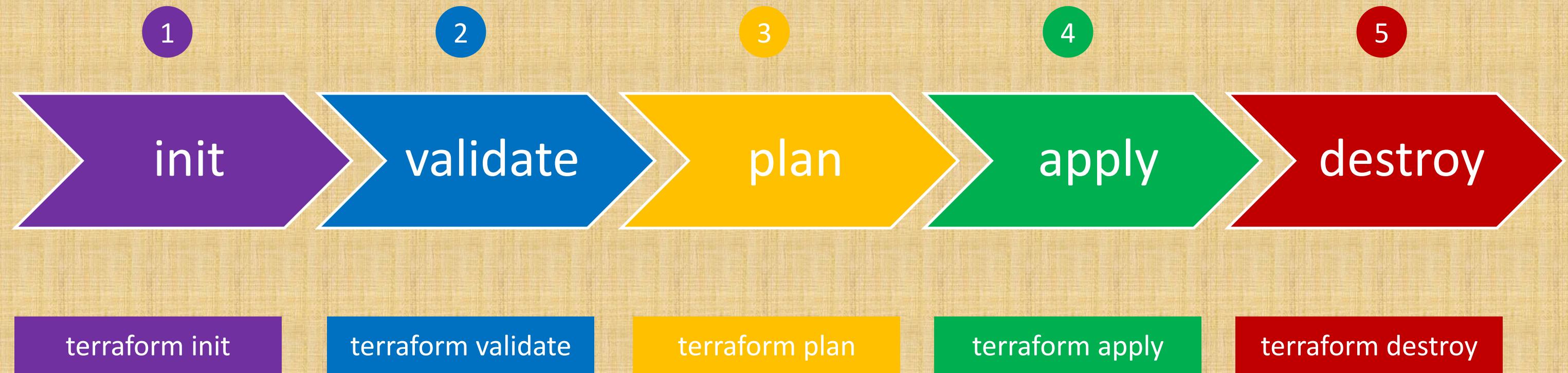
Windows OS

Linux OS

Terraform Command Basics



Terraform Workflow



Terraform Workflow

1

init

2

validate

3

plan

4

apply

5

destroy

- 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

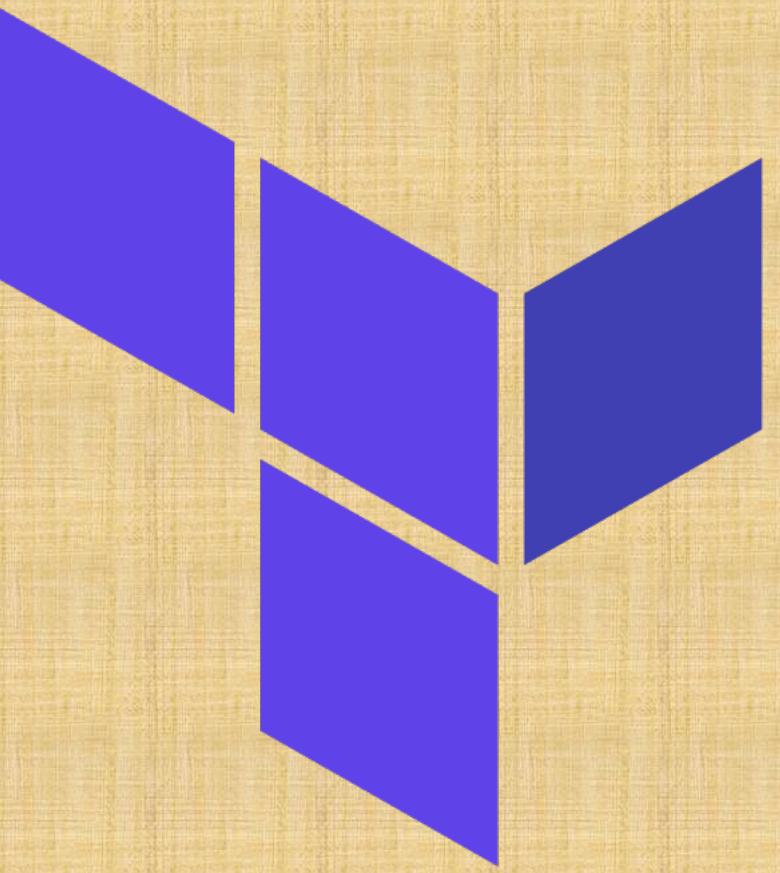
- Validates the terraform configurations files in that respective directory to ensure they are syntactically valid and internally consistent.

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

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

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

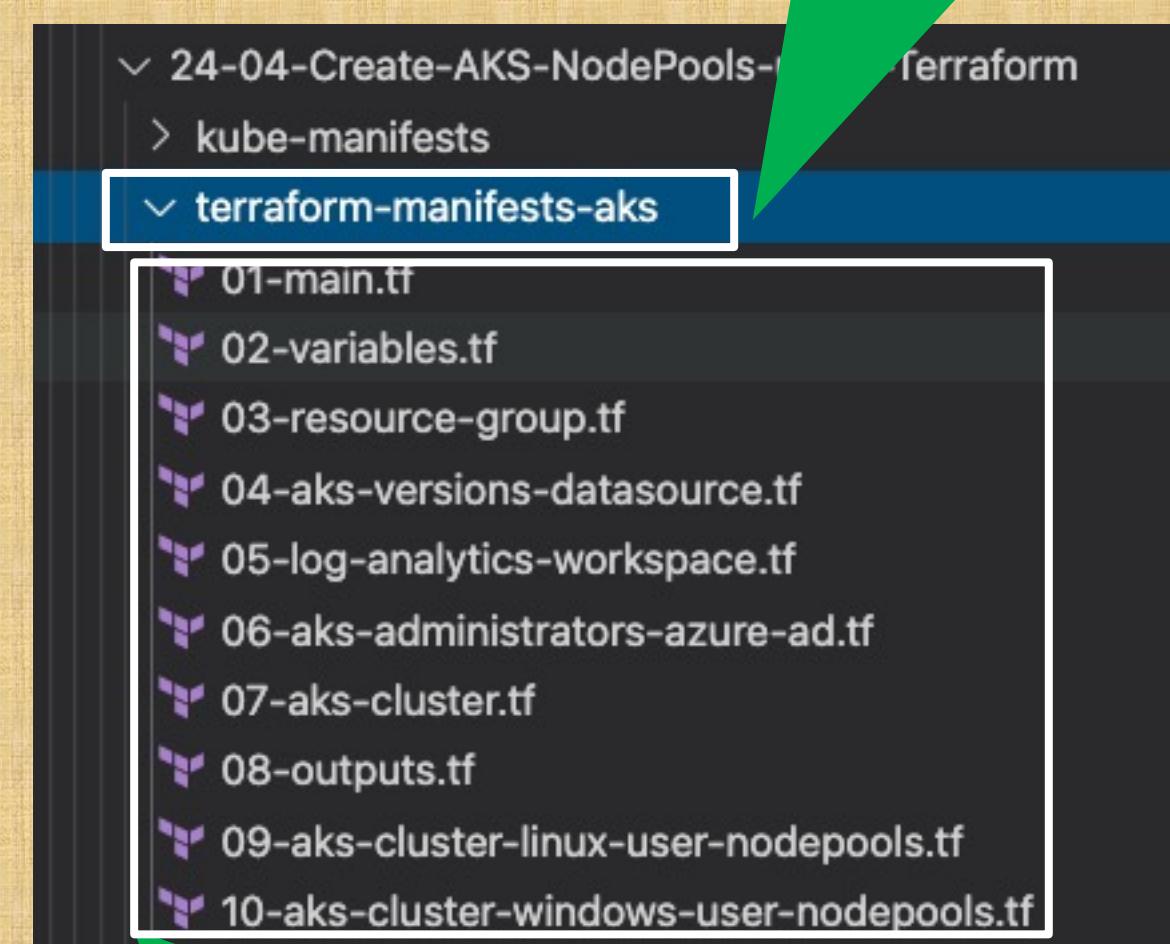
Terraform Language Basics



Terraform Language Basics – Files

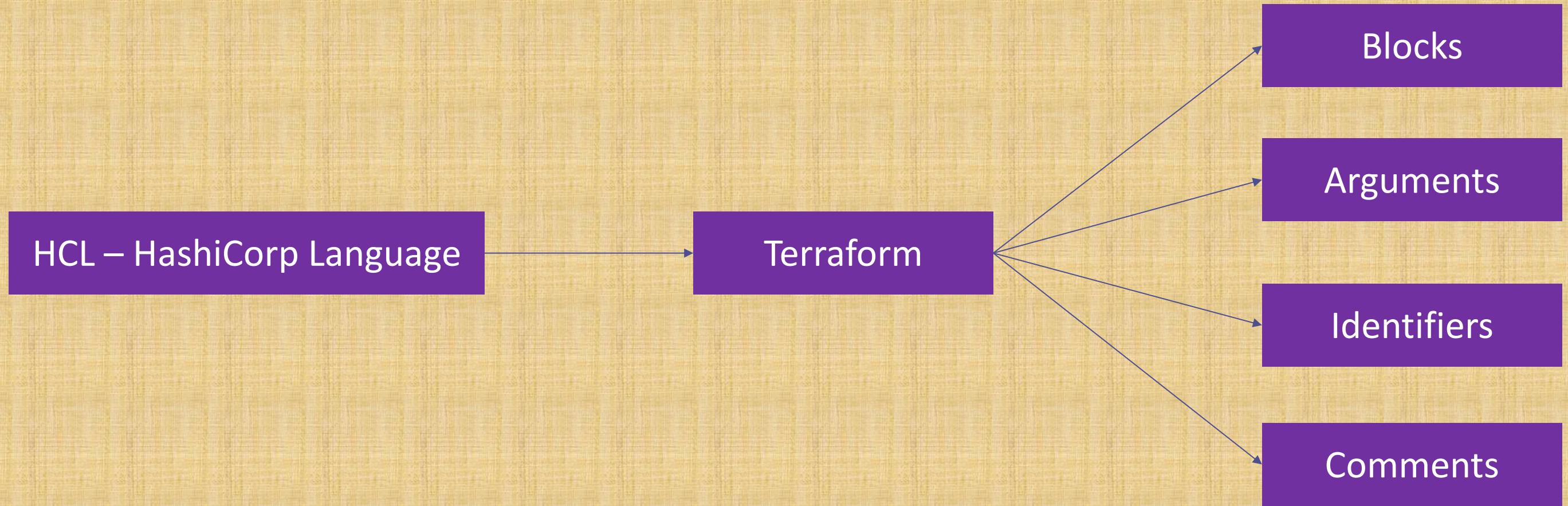
- Code in the Terraform language is stored in plain text files with the `.tf` file extension.
- There is also a JSON-based variant of the language that is named with the `.tf.json` file extension.
- We can call the files containing terraform code as **Terraform Configuration Files** or **Terraform Manifests**

Terraform Working Directory



Terraform Configuration Files ending with `.tf` as extension

Terraform Language Basics – Configuration Syntax



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 Type

Top Level &
Block inside
Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners,
resource specific blocks like tags

Arguments

Block Labels

Based on Block
Type block labels
will be 1 or 2

Example:
Resource – 2
labels

Variables – 1 label

Terraform Language Basics – Configuration Syntax

Argument
Name
[or]
Identifier

```
# 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"
```

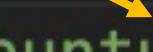
Argument
Value
[or]
Expression

Terraform Language Basics – Configuration Syntax

Single Line Comments with # or //

```
# 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
    */
}
```

Multi-line comment



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.

Terraform Block

Providers Block

Resources Block

Fundamental Blocks

Input Variables Block

Output Values Block

Local Values Block

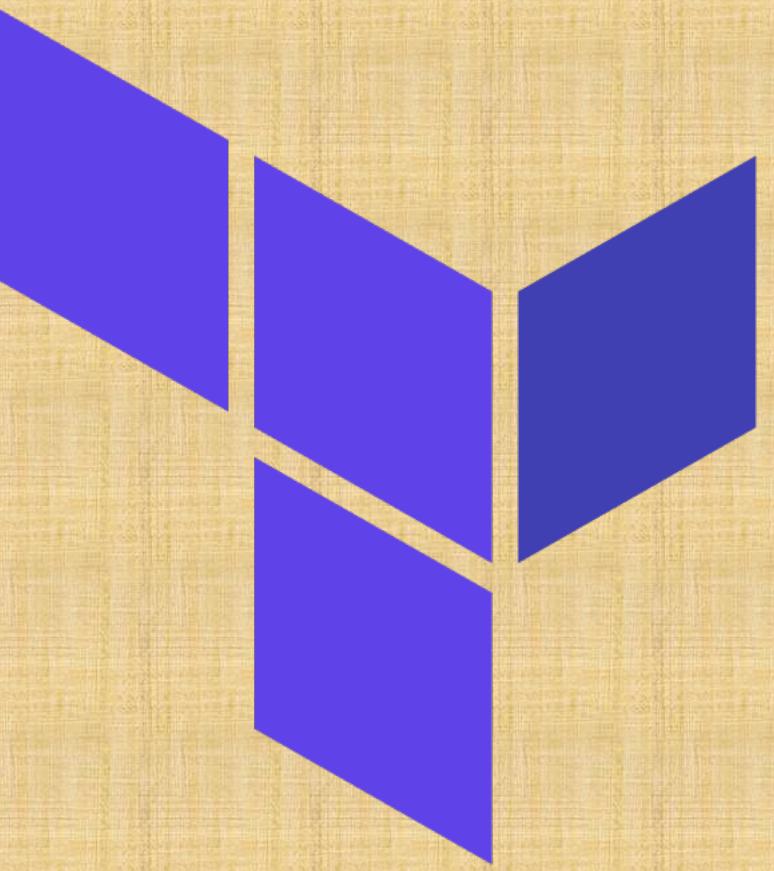
Variable Blocks

Data Sources Block

Modules Block

Calling / Referencing Blocks

Terraform Fundamental Blocks



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

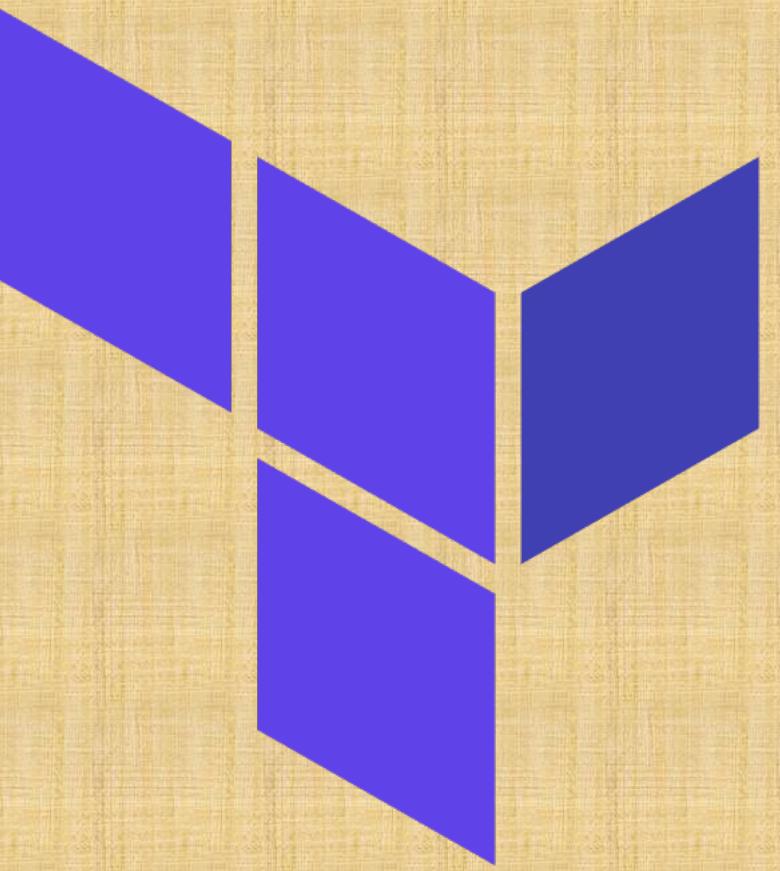
Each Resource Block describes one or more Infrastructure Objects

Resource Syntax: How to declare Resources?

Resource Behavior: How Terraform handles resource declarations?

Provisioners: We can configure Resource post-creation actions

Terraform Block



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.

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"
}
```

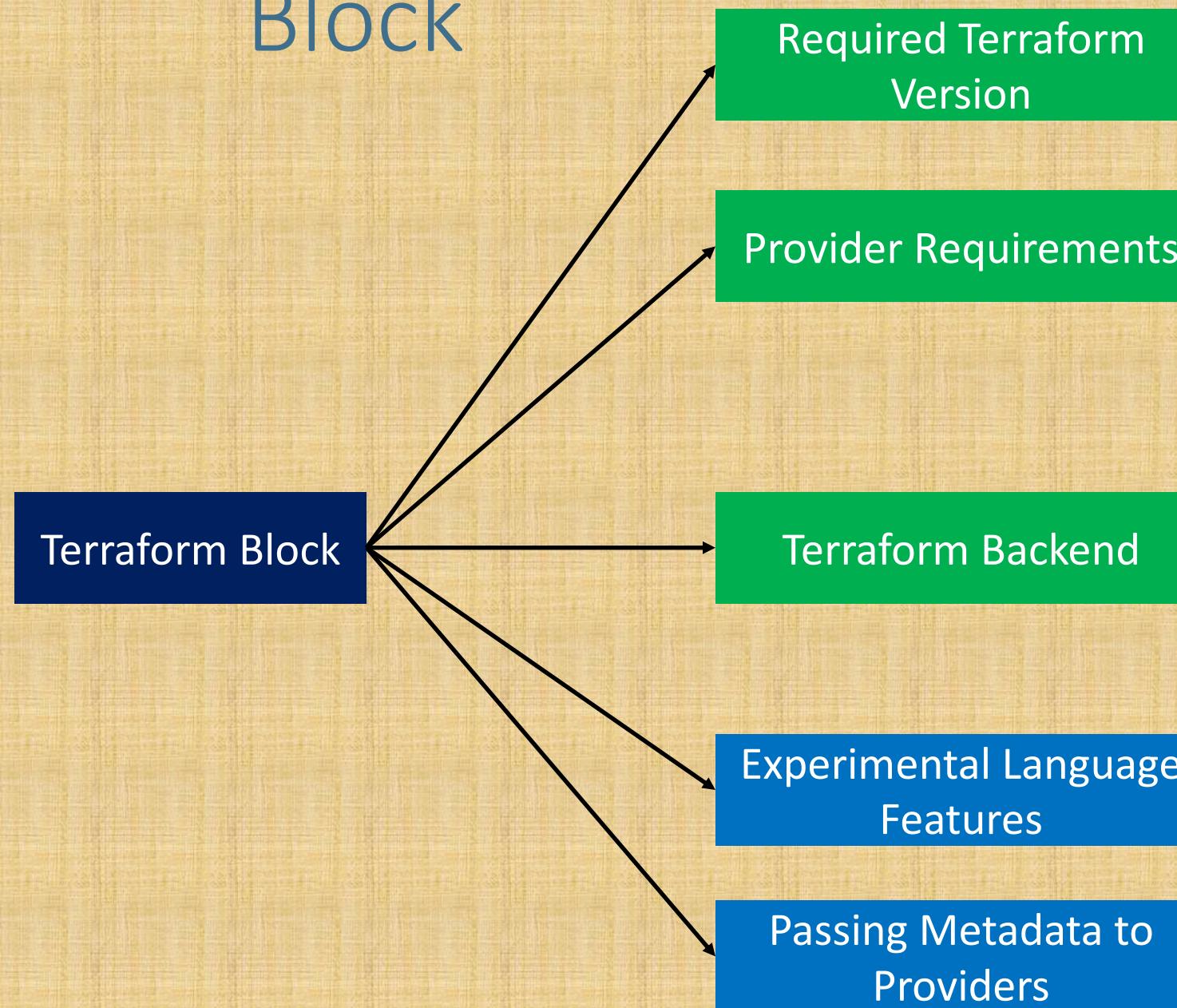
Terraform 0.13 and later:

```
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"
}
```

Terraform Block



```
terraform {
  # Required Terraform Version
  required_version = "~> 0.14.3"

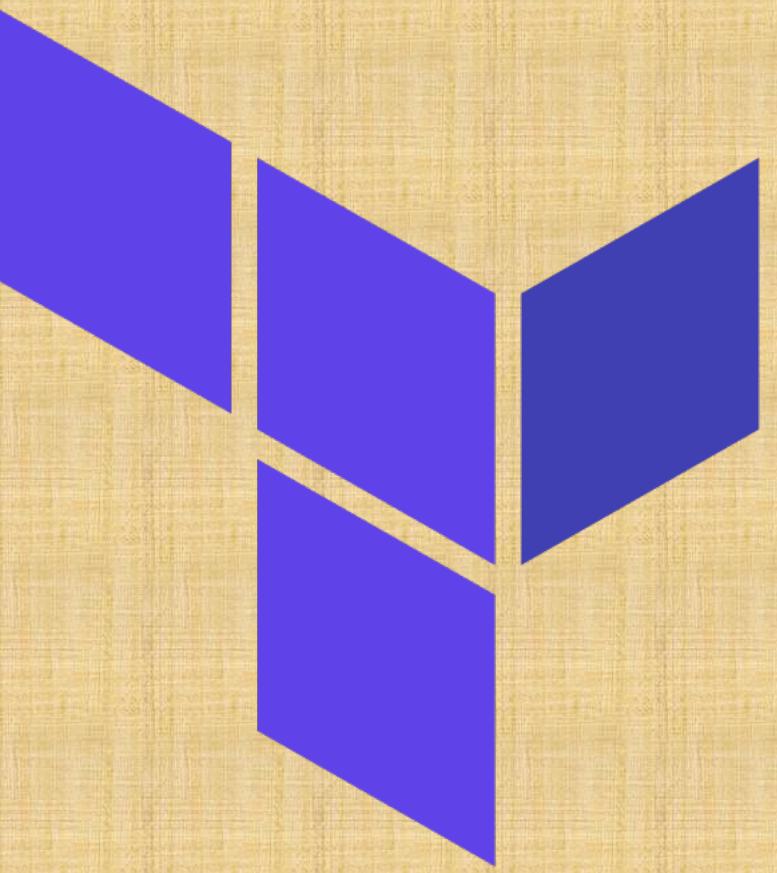
  # Required Providers and their Versions
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.21" # Optional but recommended
    }
  }

  # 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 ]

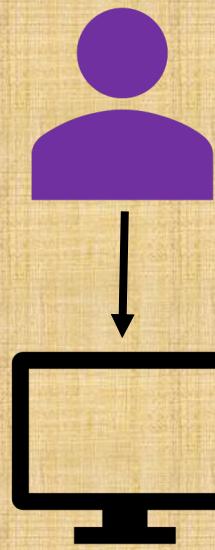
  # Passing Metadata to Providers (Super Advanced – Terraform 0.12+)
  provider_meta "my-provider" {
    hello = "world"
  }
}
```

Terraform Providers



Terraform Providers

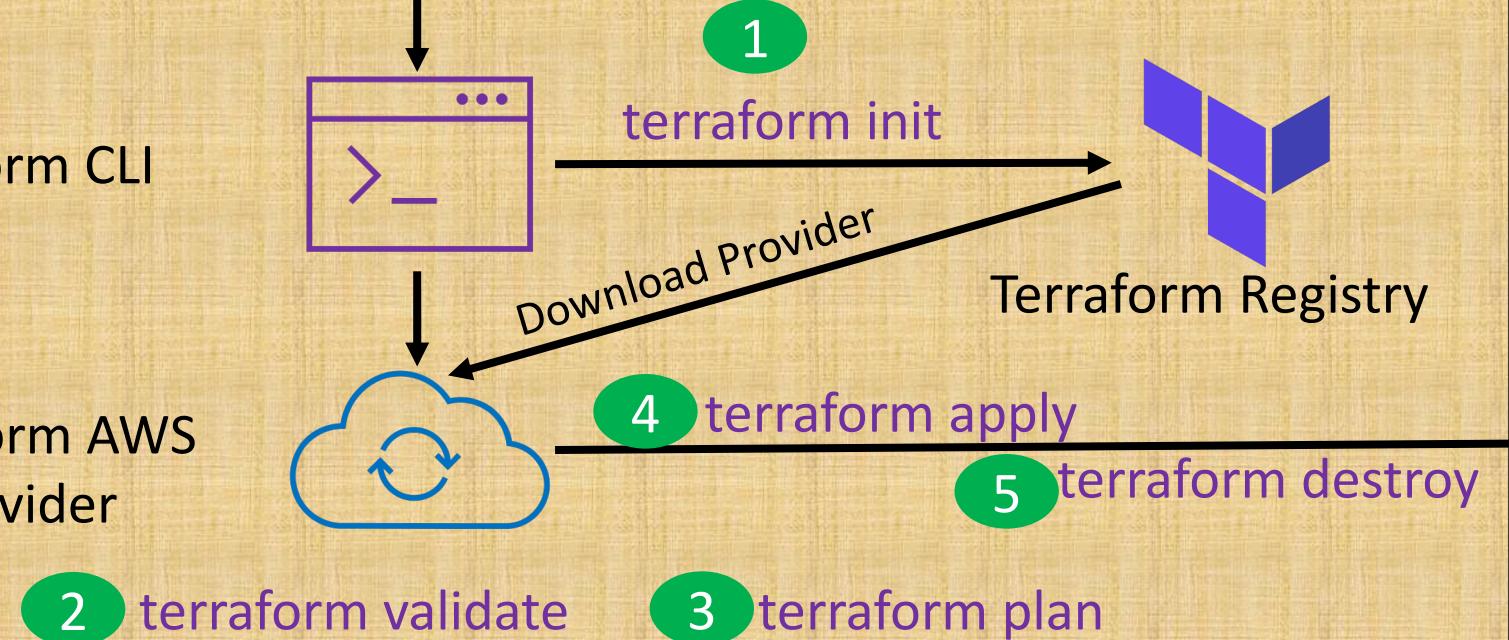
Terraform Admin



Local Desktop

Terraform CLI

Terraform AWS Provider



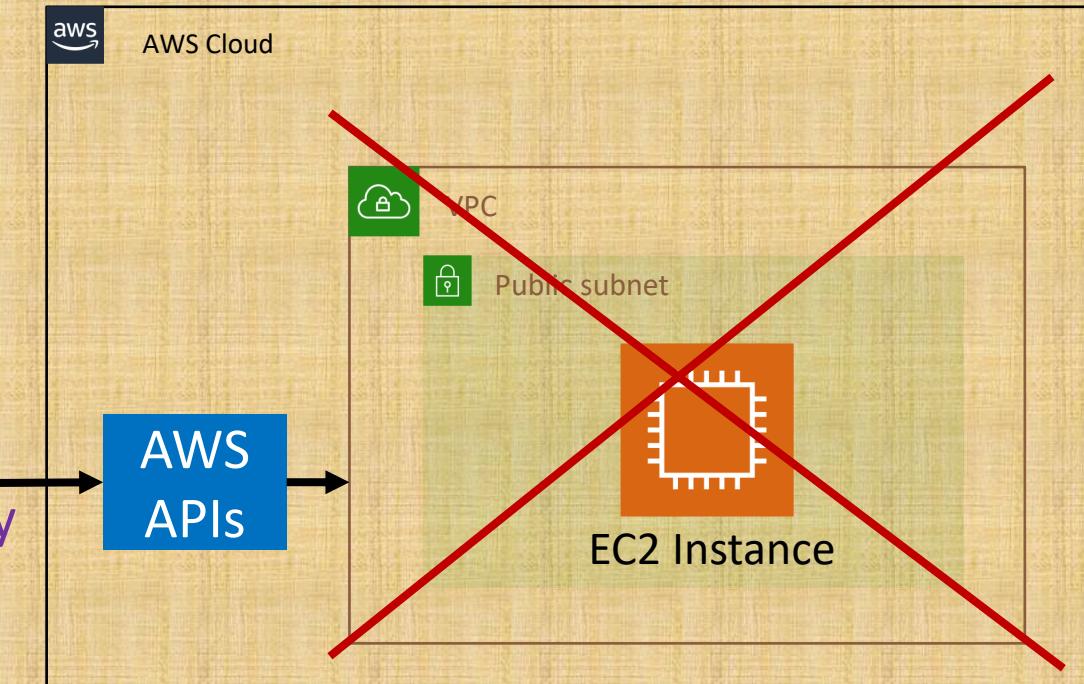
Providers are **HEART** of Terraform

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



Provider Requirements

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

Terraform Providers

Provider Configuration

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

Dependency Lock File

```
└─ terraform-manifests
    └─ .terraform
        └─ .terraform.lock.hcl
    └─ ec2-instance.tf
    └─ terraform.tfstate
    └─ terraform.tfstate.backup

# 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/Tz8zv1zb78ZaiyJk0OMGIViZwbYrLuQk3kojPM91c=",
    "zh:4a9a66caf1964cd3b61fb3eb0da417195a529ccb8e496f266b0778335d11c8",
    "zh:514f2f006ae68db715d86781673faf9483292deab235c7402ff306e0e92ea11a",
    "zh:52777b61109fddb9011728f6650ef01a639a0590aeffe34ed7de7ba10d0c31803",
    "zh:67784dc8c8375ab37103eea1258c3334ee92be6de033c2b37e3a2a65d0005142",
    "zh:76d4c8be2ca4a3294fb51fb58de1fe03361d3bc403820270cc8e71a04c5fa006",
    "zh:8f900b1cfdf6e8fb1a9d0382ecaa5056a3a84c94e313fb9e92c89de271cdede",
    "zh:d0ac346519d0df124df89be2d803eb53f373434890f6ee3fb27112802f9eac59",
    "zh:d6256feedada82cbfb3b1dd6dd9ad02048f23120ab50e6146a541cb11a108cc1",
    "zh:db2fe0d2e77c02e9a74e1ed694aa352295a50283f9a1cf896e5be252af14e9f4",
    "zh:eda61e889b579bd90046939a5b40cf5dc9031fb5a819fc3e4667a78bd432bdb2",
  ]
}
```

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 = [
```

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

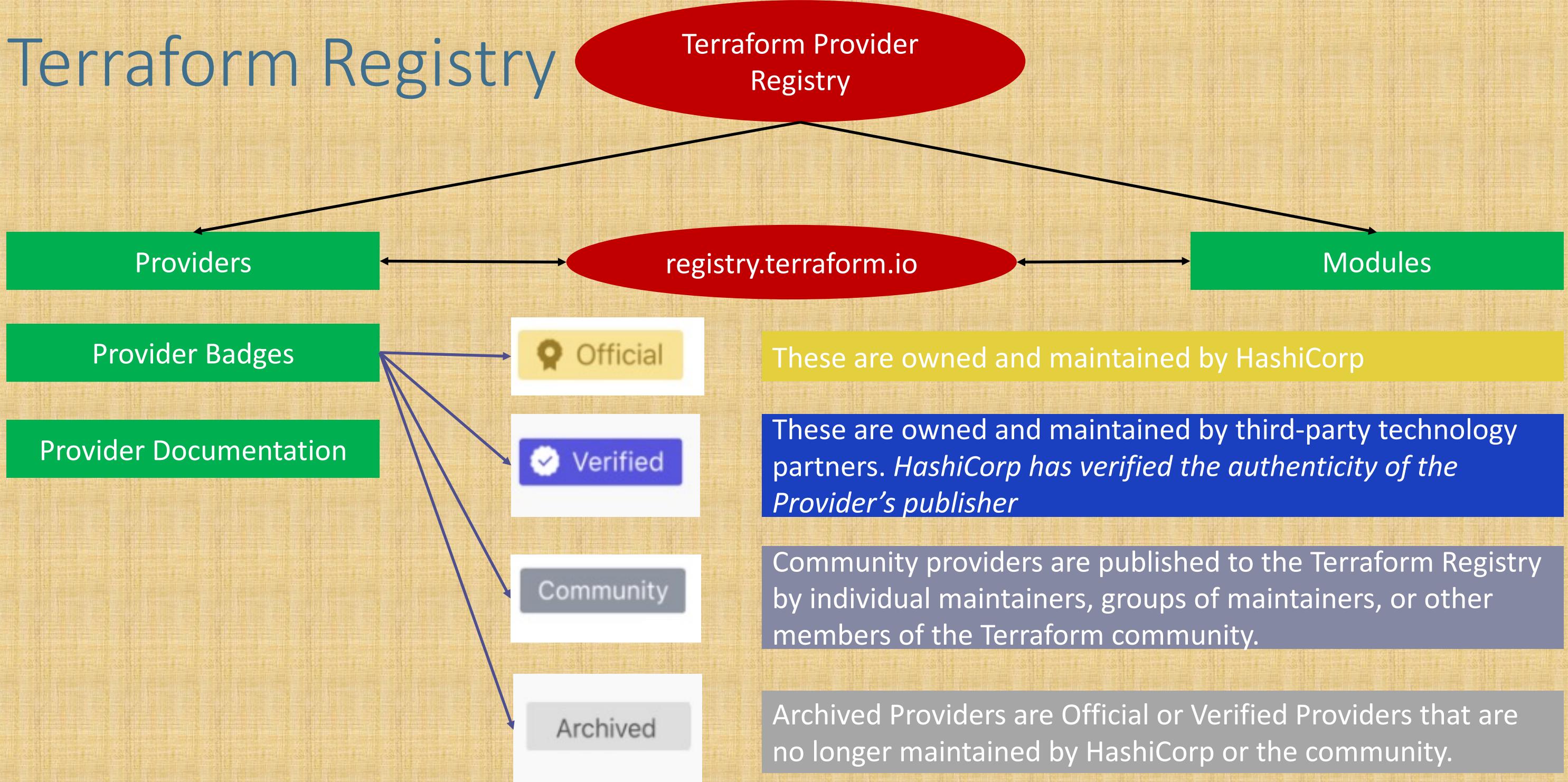
Source addresses consist of **three parts** delimited by **slashes (/)**

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

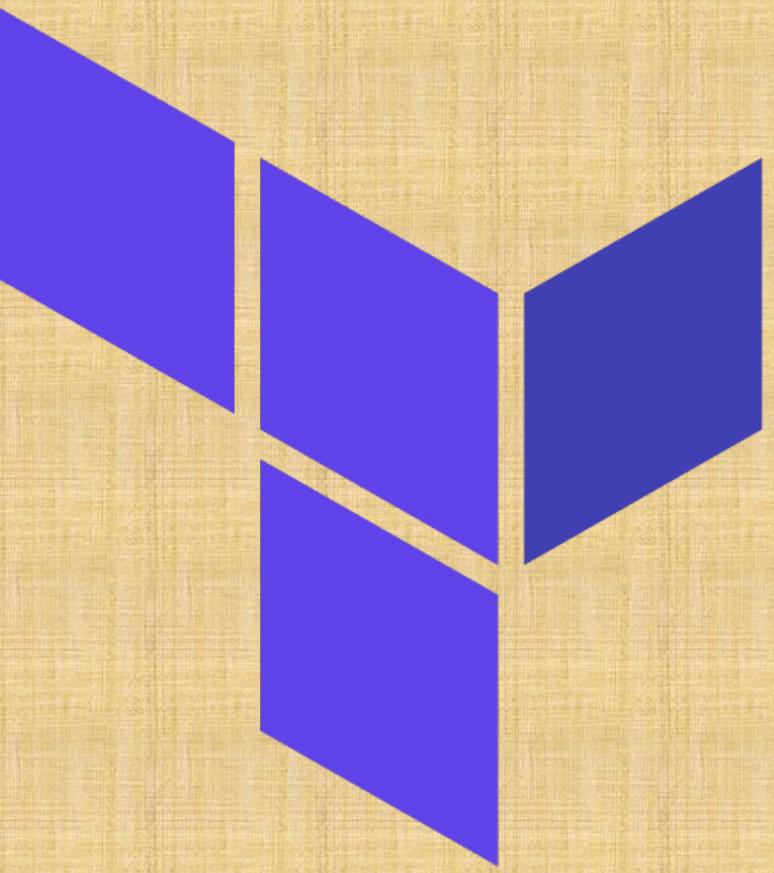
registry.terraform.io/hashicorp/aws

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

Terraform Registry



Terraform Multiple Providers



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**

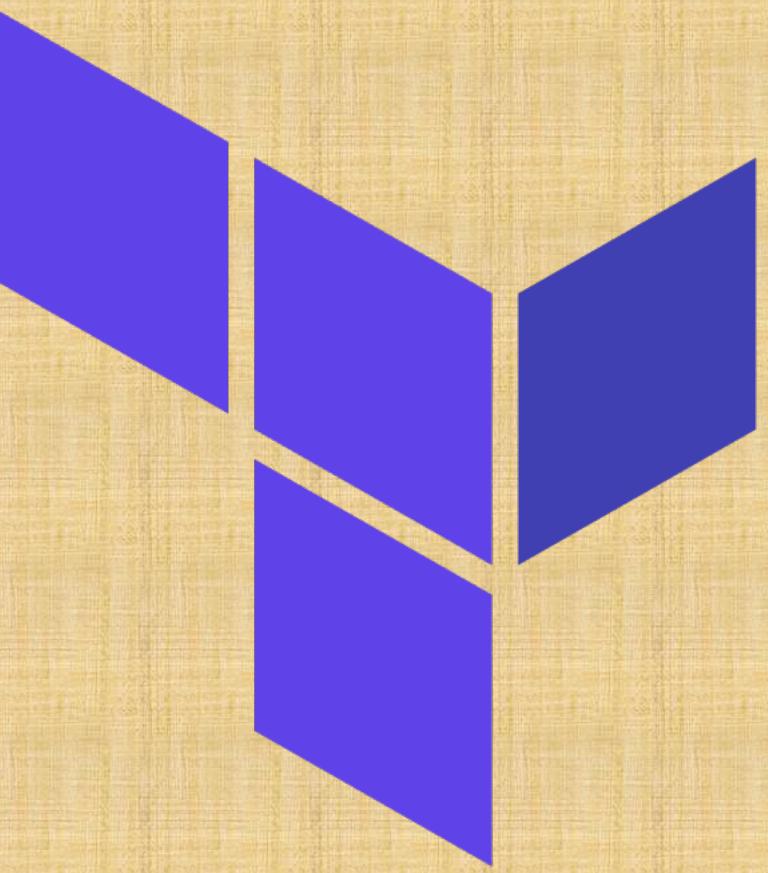
We can **use** the alternate provider in a resource, data or module by referencing it as **<PROVIDER NAME>.<ALIAS>**

```
# 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"
  }
}
```

Terraform Dependency Lock File



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

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 = [
```

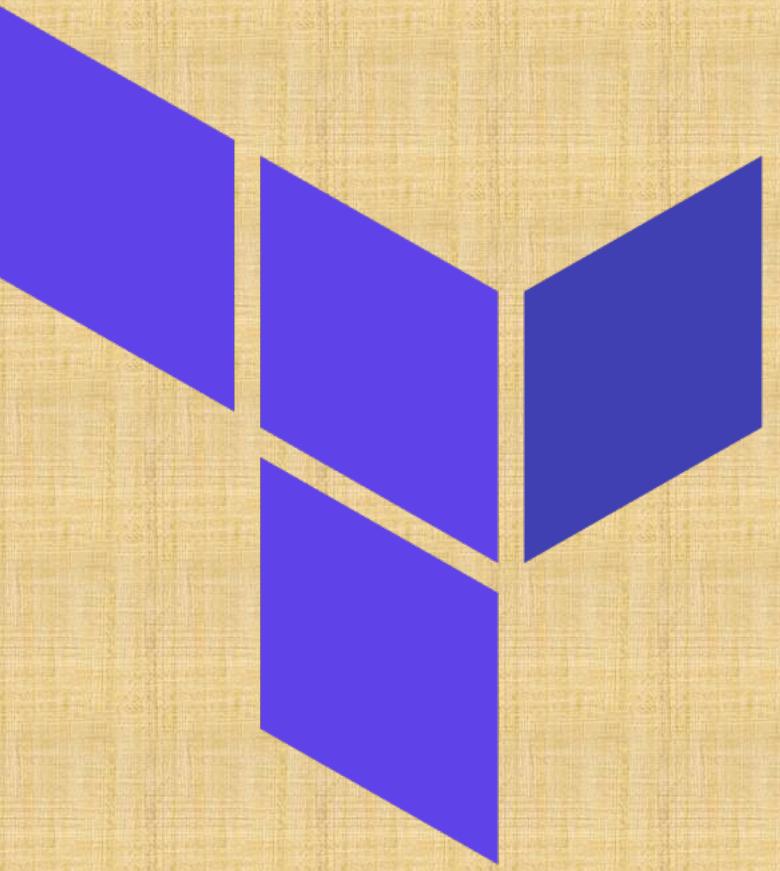
Importance of Dependency Lock File

Provider	Version Constraint	terraform init (no lock file)	terraform init (lock file)
aws	<code>>= 2.0</code>	Latest version (3.18.0)	Lock file version (2.50.0)
random	<code>3.0.0</code>	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**.

Terraform Resources Introduction



Terraform Resources

Terraform
Language Basics

Terraform
Resource Syntax

Terraform
Resource Behavior

Terraform
State

Resource
Meta-Argument
count

Resource
Meta-Argument
depends_on

Resource
Meta-Argument
for_each

Resource
Meta-Argument
lifecycle

Terraform Resources - Duration

^ Terraform Resources

26 lectures • 3hr 1min

WHY?

- ▶ Step-01: Terraform Resources Syntax Introduction 10:35
- ▶ Step-02: Terraform Resources Behavior Part-1 06:56
- ▶ Step-03: Terraform Resources Behavior Part-2 13:11
- ▶ Step-04: Understand Terraform State and Review terraform.tfstate file 07:46
- ▶ Step-05: Understand Terraform Resource Behavior Update-In-Place 04:22
- ▶ Step-06: Understand Terraform Resource Behavior Destroy-and-Recreate-and-also-Delete 04:53
- ▶ Step-07: Terraform Desired & Current State Highlevel and Clean-Up 05:02

Language Syntax

Resource Syntax

Resource Behavior

State Fundamentals

Resource Meta-Arguments

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 Type

Top Level &
Block inside
Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners,
resource specific blocks like tags

Arguments

Block Labels

Based on Block
Type block labels
will be 1 or 2

Example:
Resource – 2
labels

Variables – 1 label

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

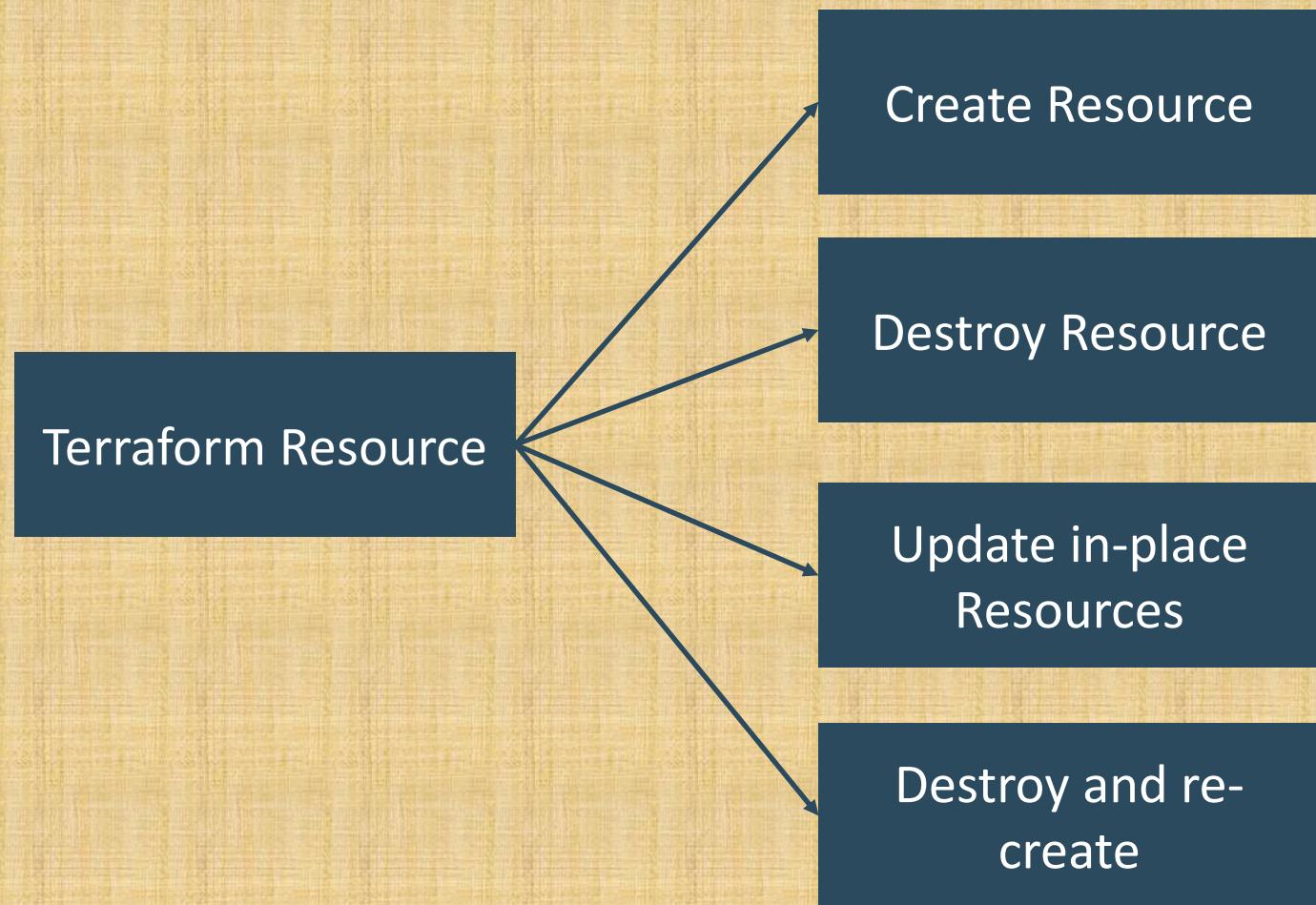
Meta-Arguments: Can be used with any resource to change the behavior of resources

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"
  }
}
```

Resource Behavior



Create resources that exist in the configuration but are **not associated** with a real infrastructure object in the state.

Destroy resources that **exist in the state** but no longer exist in the configuration.

Update **in-place resources** whose arguments have changed.

Destroy and re-create resources whose arguments have changed but which **cannot be updated in-place** due to remote API limitations.

Terraform State

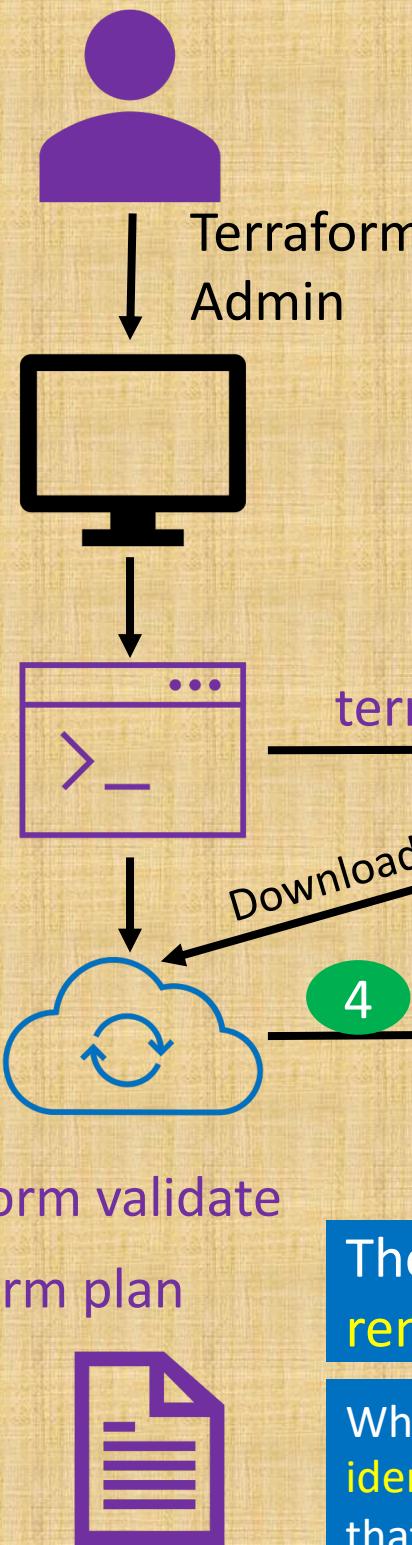
Terraform State

Local Desktop

Terraform CLI

Terraform AWS Provider

Terraform State File
`terraform.tfstate`



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.

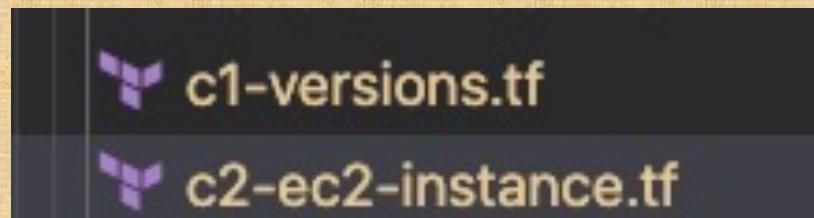
This state is stored by default in a local file named "**terraform.tfstate**", but it can also be stored **remotely**, which works better in a **team** environment.

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.

Desired & Current Terraform States

Terraform Configuration Files



Real World Resource – EC2 Instance

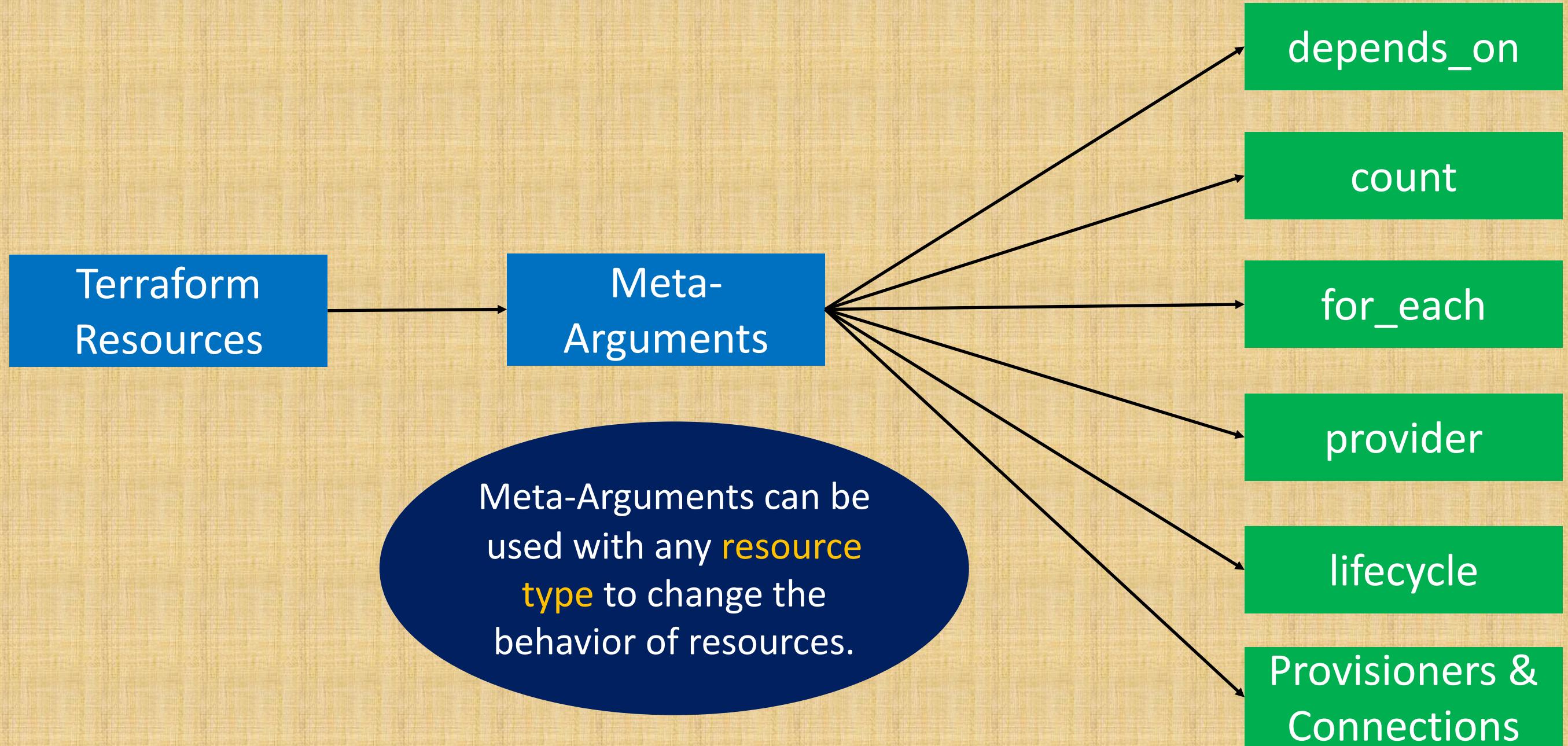
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
web	i-0663449fef49e9cf5	Running	t2.micro	2/2 checks ...	No alarms +	us-east-1b
Instance: i-0663449fef49e9cf5 (web)						
Details	Security	Networking	Storage	Status checks	Monitoring	Tags
Instance ID i-0663449fef49e9cf5 (web)	Public IPv4 address 54.144.73.100 open address	Private IPv4 addresses 172.31.94.137	Public IPv4 DNS ec2-54-144-73-100.compute-1.amazonaws.com open address	Private IPv4 DNS ip-172-31-94-137.ec2.internal	VPC ID vpc-54972d2e (default-vpc)	Subnet ID subnet-d2e590fc
Instance state Running	Elastic IP addresses -	AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations.	IAM Role -			

Desired State



Current State

Resource Meta-Arguments



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 ]
}
```

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

Buckets (30)	
Buckets are containers for data stored in S3. Learn more	
<input type="text"/> my	
Name	AWS Region
dev-my-dapp-bucket	US East (N. Virginia) us-east-1
prod-my-papp-bucket	US East (N. Virginia) us-east-1
qa-my-qapp-bucket	US East (N. Virginia) us-east-1
stag-my-sapp-bucket	US East (N. Virginia) us-east-1

```
# 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}"  
}
```

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
}
```



The screenshot shows the AWS IAM User Management console interface. At the top, there are buttons for 'Add user' (blue) and 'Delete user' (red). On the right, there are three small icons: a refresh symbol, a gear, and a question mark. Below the buttons is a search bar with the placeholder 'Find users by username or access key'. To the right of the search bar, it says 'Showing 7 results'. The main area is a table with the following columns: 'User name', 'Groups', 'Access key age', 'Password age', 'Last activity', and 'MFA'. There are seven rows, each corresponding to one of the users created in the Terraform code: TMadhu, TJames, TJack, and TDave. Each row has a checkbox in the first column and the user's name in the second column. The other columns show 'None' for all users.

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

Resource Meta-Argument lifecycle

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

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

create_before_destroy

prevent_destroy

ignore_changes

```
# 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
  }
}
```

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

```
# 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,
    ]
  }
}
```

Github Step-by-Step Documentation

The screenshot shows a GitHub repository page for the 'hashicorp-certified-terraform-associate' repository, specifically the 'master' branch. The URL in the address bar is github.com/stacksimplify/hashicorp-certified-terraform-associate/tree/master/04-Terraform-Resources. The page title is 'hashicorp-certified-terraform-associate / 04-Terraform-Resources /'. A user profile icon for 'stacksimplify' is visible, followed by the text 'Welcome to StackSimplify'. Below this, there is a list of five folder entries under the heading '04-Terraform-Resources /':

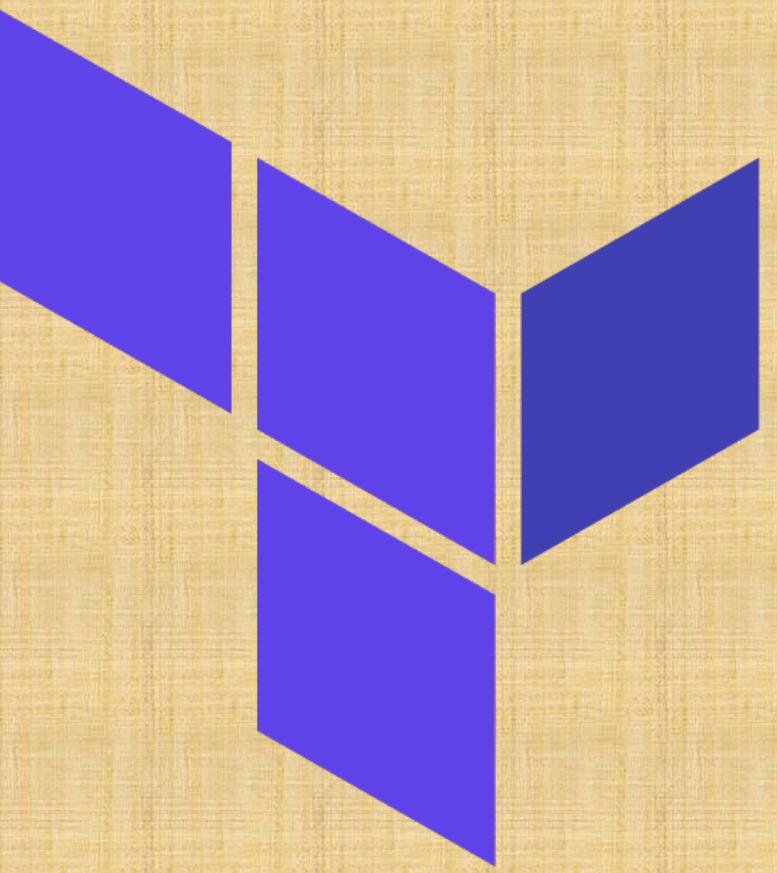
Folder	Content
04-01-Resource-Syntax-and-Behavior	Welcome to StackSimplify
04-02-Meta-Argument-depends_on	Welcome to Stack Simplify
04-03-Meta-Argument-count	Welcome to Stack Simplify
04-04-Meta-Argument-for_each	Welcome to Stack Simplify
04-05-Meta-Argument-lifecycle	Welcome to Stack Simplify

Practical Examples For Each Concept

```
└─ 04-Terraform-Resources
    └─ 04-01-Resource-Syntax-and-Behavior
        └─ terraform-manifests
            └─ c1-versions.tf
            └─ c2-ec2-instance.tf
        └─ README.md
    └─ 04-02-Meta-Argument-depends_on
        └─ terraform-manifests
            └─ apache-install.sh
            └─ c1-versions.tf
            └─ c2-vpc.tf
            └─ c3-ec2-instance.tf
            └─ c4-elastic-ip.tf
        └─ README.md
    └─ 04-03-Meta-Argument-count
        └─ terraform-manifests
            └─ c1-versions.tf
            └─ c2-ec2-instance.tf
        └─ README.md
```

```
└─ 04-04-Meta-Argument-for_each
    └─ v1-for_each-maps
        └─ c1-versions.tf
        └─ c2-s3bucket.tf
    └─ v2-for_each-toset
        └─ c1-versions.tf
        └─ c2-iamuser.tf
    └─ README.md
    └─ 04-05-Meta-Argument-lifecycle
        └─ v1-create_before_destroy
            └─ c1-versions.tf
            └─ c2-ec2-instance.tf
        └─ v2-prevent_destroy
            └─ c1-versions.tf
            └─ c2-ec2-instance.tf
        └─ v3-ignore_changes
            └─ c1-versions.tf
            └─ c2-ec2-instance.tf
```

Terraform Resource Syntax



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 Type

Top Level &
Block inside
Blocks

Top Level Blocks: resource, provider

Block Inside Block: provisioners,
resource specific blocks like tags

Arguments

Block Labels

Based on Block
Type block labels
will be 1 or 2

Example:
Resource – 2
labels

Variables – 1 label

Terraform Language Basics – Configuration Syntax

Argument
Name
[or]
Identifier

```
# 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
Value
[or]
Expression

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

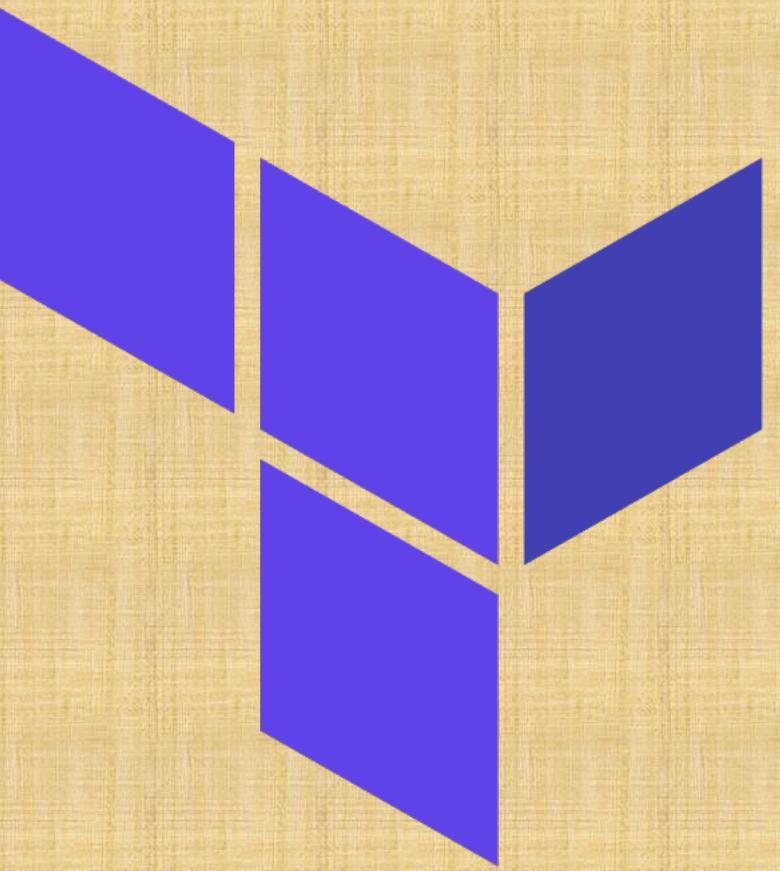
Resource Arguments: Will be specific to resource type. Argument Values can make use of **Expressions** or other Terraform **Dynamic Language Features**

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

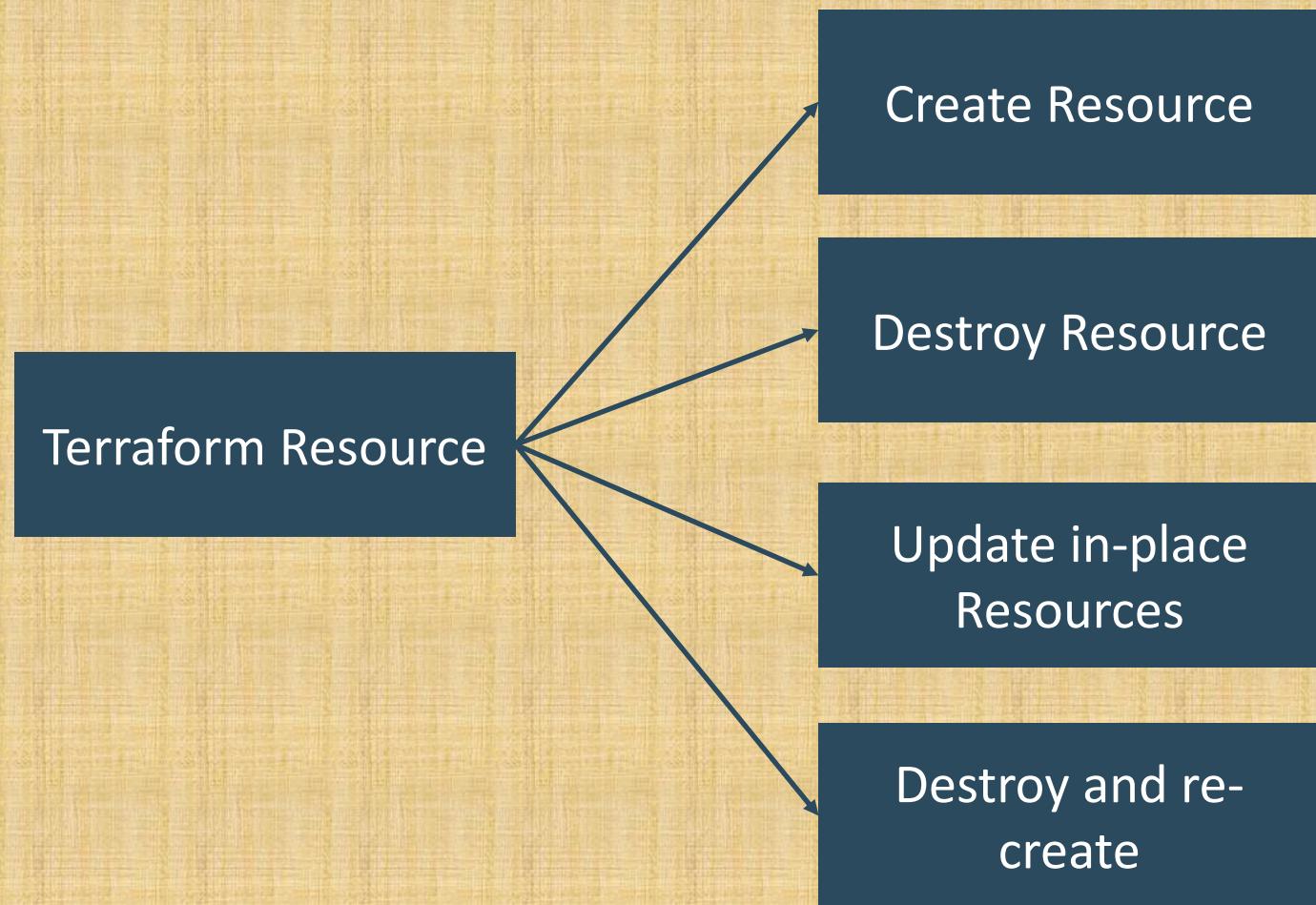
# Resource Block to Create VPC
resource "aws_vpc" "vpc_us-west-1" {
  provider = aws.aws-west-1
  cidr_block = "10.2.0.0/16"
  tags = {
    "Name" = "vpc-1"
  }
}
```

Terraform

Resource Behavior

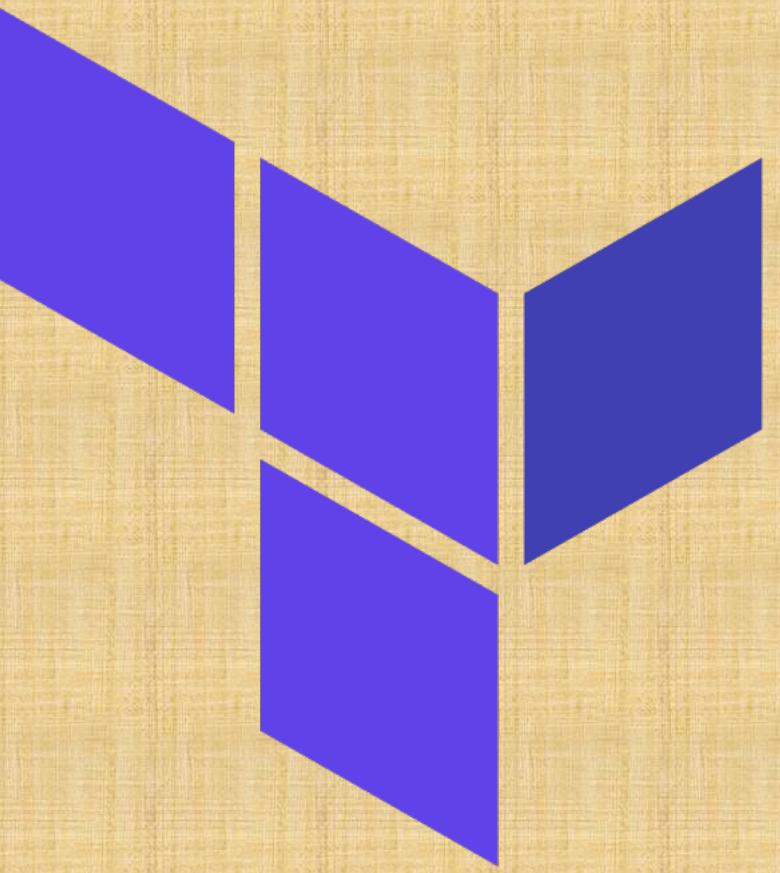


Resource Behavior



Terraform State

Terraform State



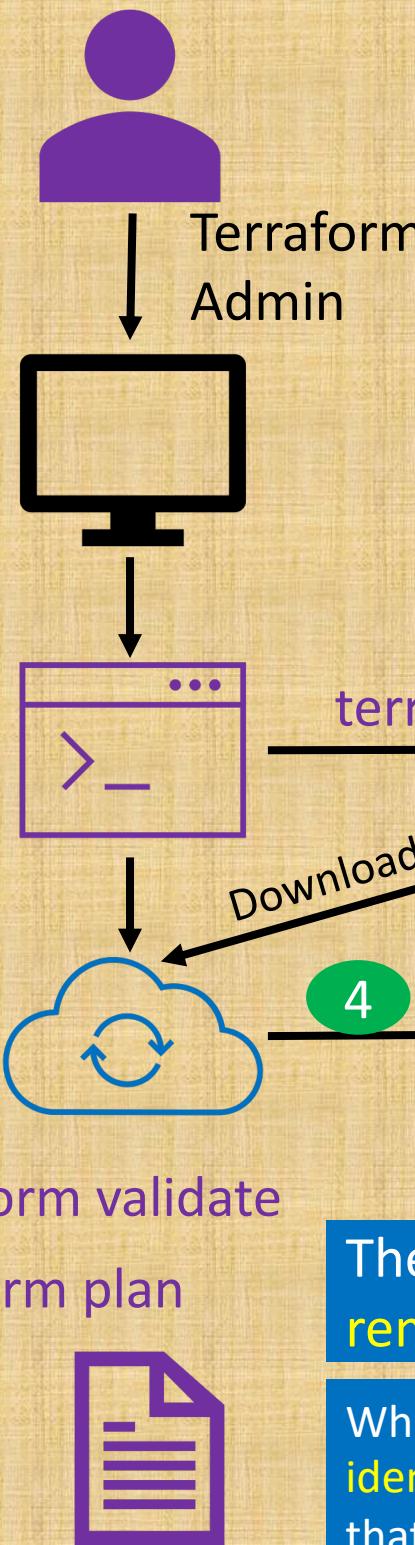
Terraform State

Local Desktop

Terraform CLI

Terraform AWS Provider

Terraform State File
`terraform.tfstate`



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.

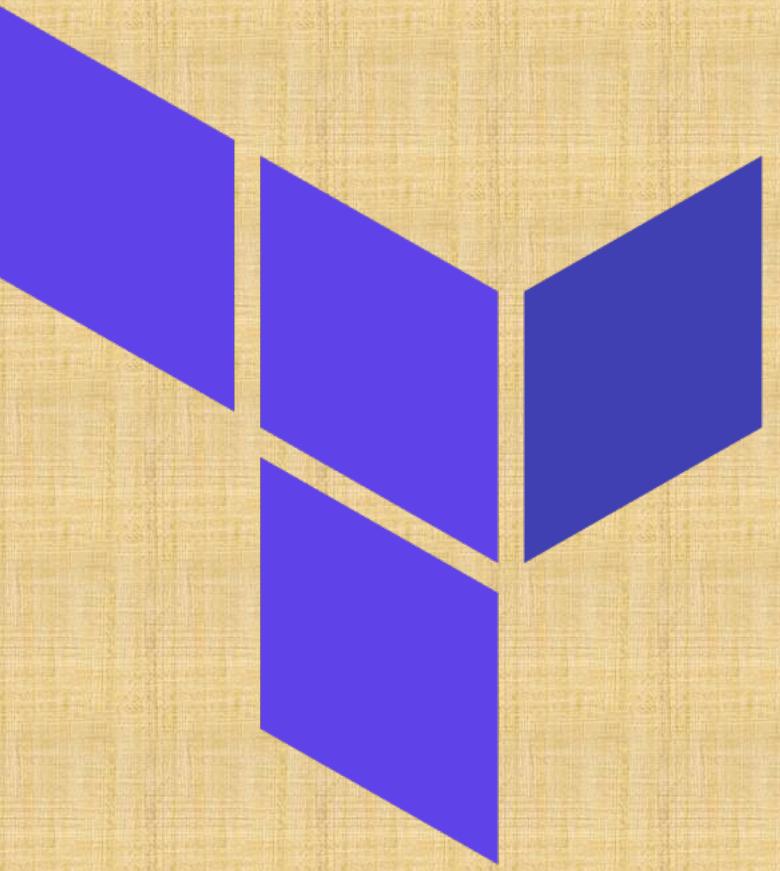
The **primary purpose** of Terraform state is to store **bindings** between objects in a **remote system** and resource instances **declared** in your configuration.

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.

Terraform

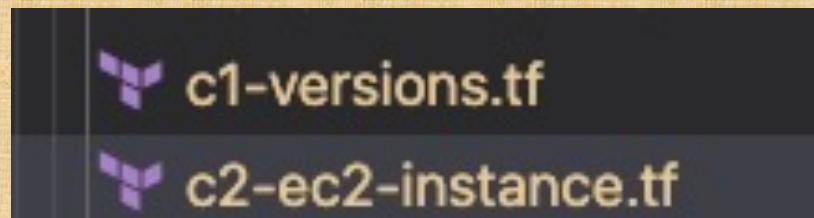
State

Desired & Current



Desired & Current Terraform States

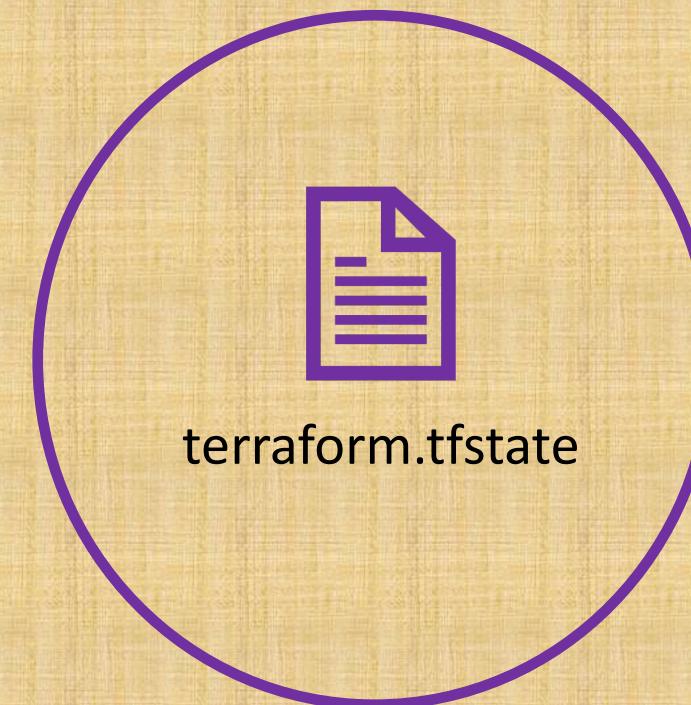
Terraform Configuration Files



Real World Resource – EC2 Instance

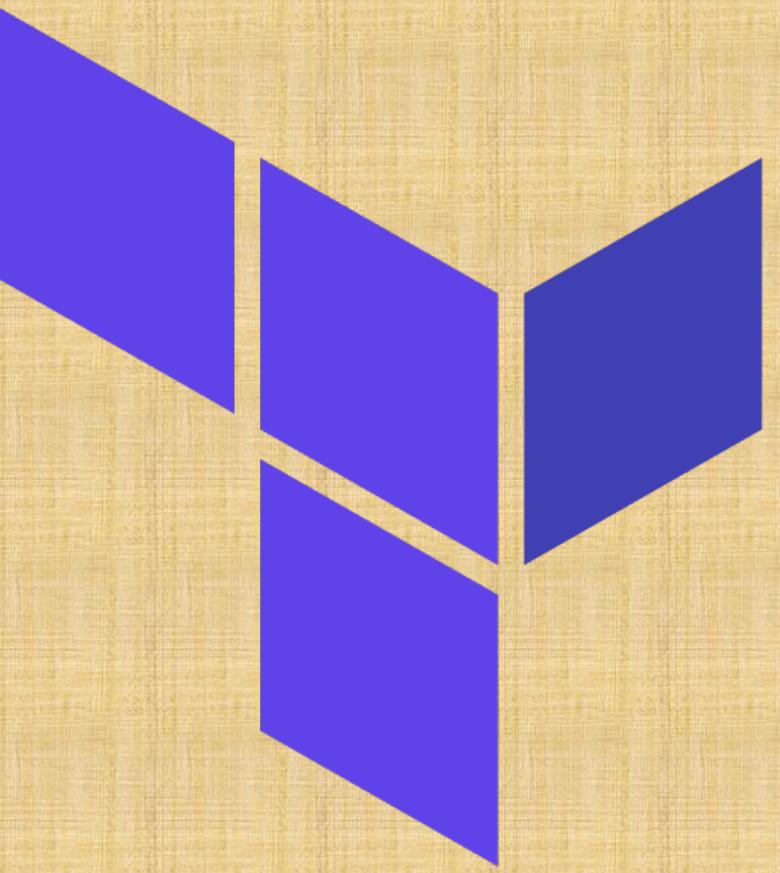
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone
web	i-0663449fef49e9cf5	Running	t2.micro	2/2 checks ...	No alarms +	us-east-1b
Instance: i-0663449fef49e9cf5 (web)						
Details	Security	Networking	Storage	Status checks	Monitoring	Tags
Instance ID i-0663449fef49e9cf5 (web)	Public IPv4 address 54.144.73.100 open address	Private IPv4 addresses 172.31.94.137	Public IPv4 DNS ec2-54-144-73-100.compute-1.amazonaws.com open address	Private IPv4 DNS ip-172-31-94-137.ec2.internal	VPC ID vpc-54972d2e (default-vpc)	Subnet ID subnet-d2e590fc
Instance state Running	Elastic IP addresses -	AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations.	IAM Role -			

Desired State

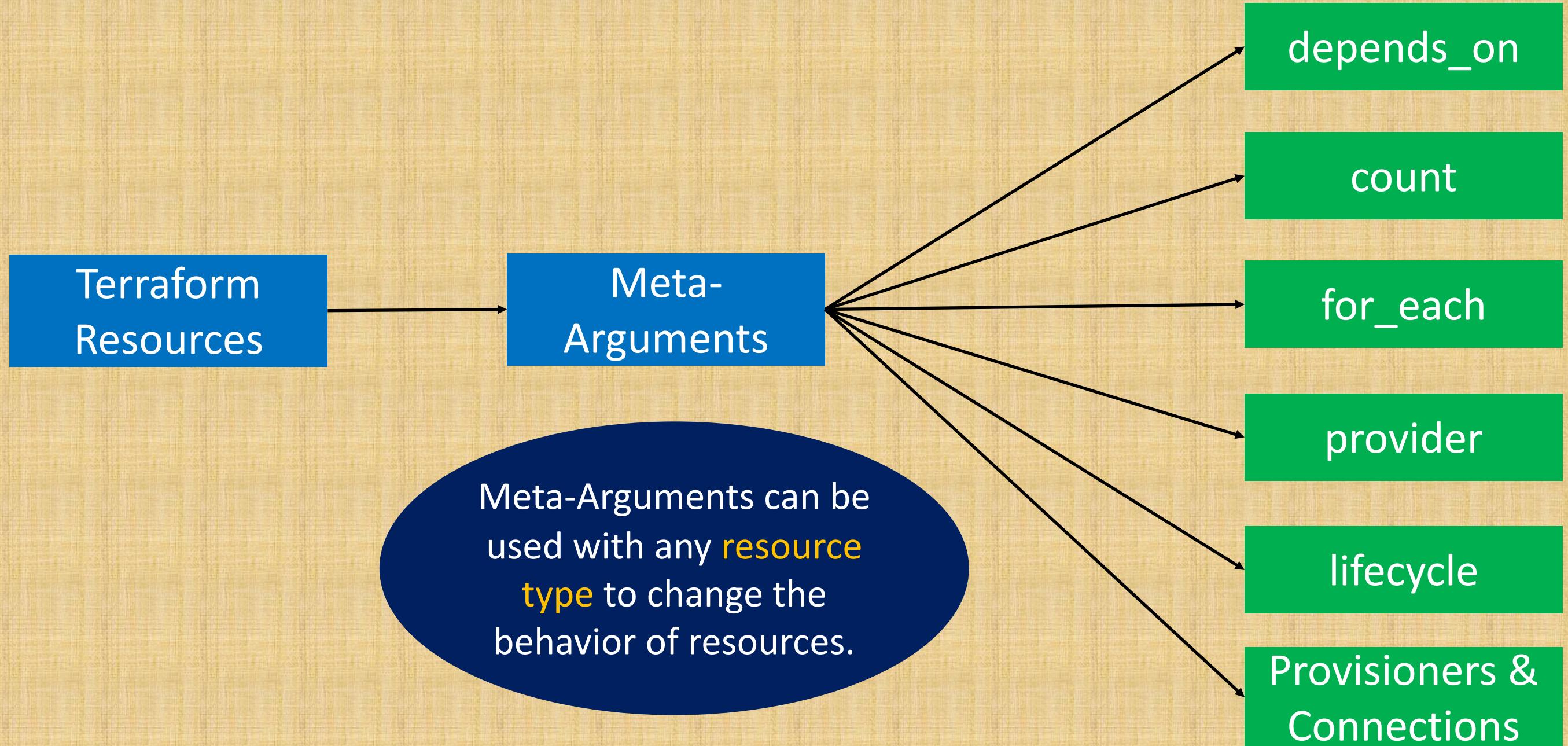


Current State

Terraform Resource Meta-Arguments



Resource Meta-Arguments



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

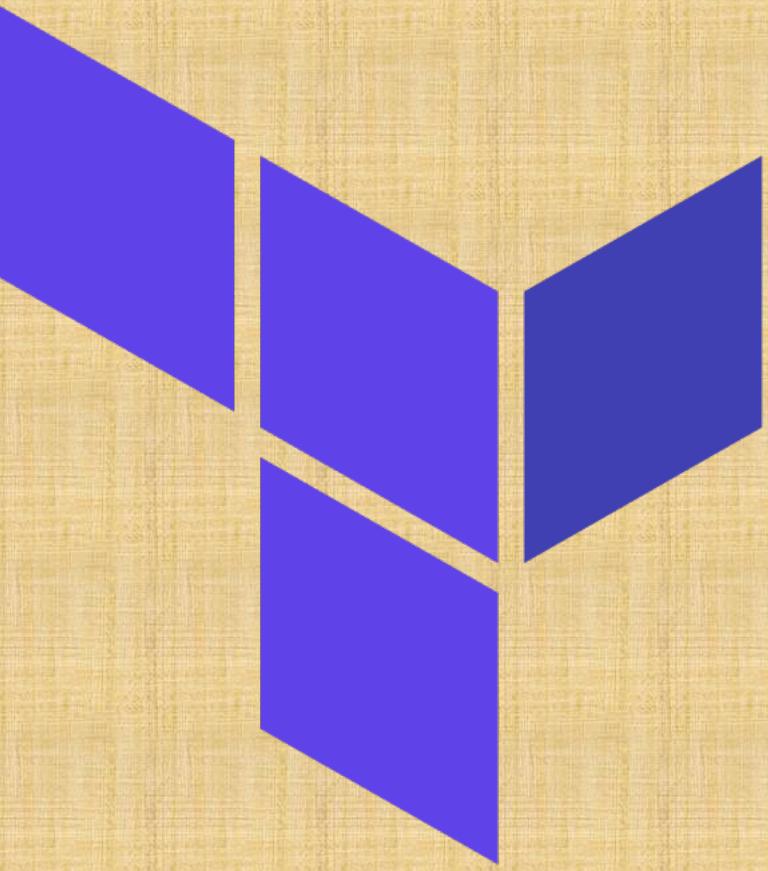
lifecycle

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**)

Terraform Resource Meta-Argument depends_on



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.

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
```

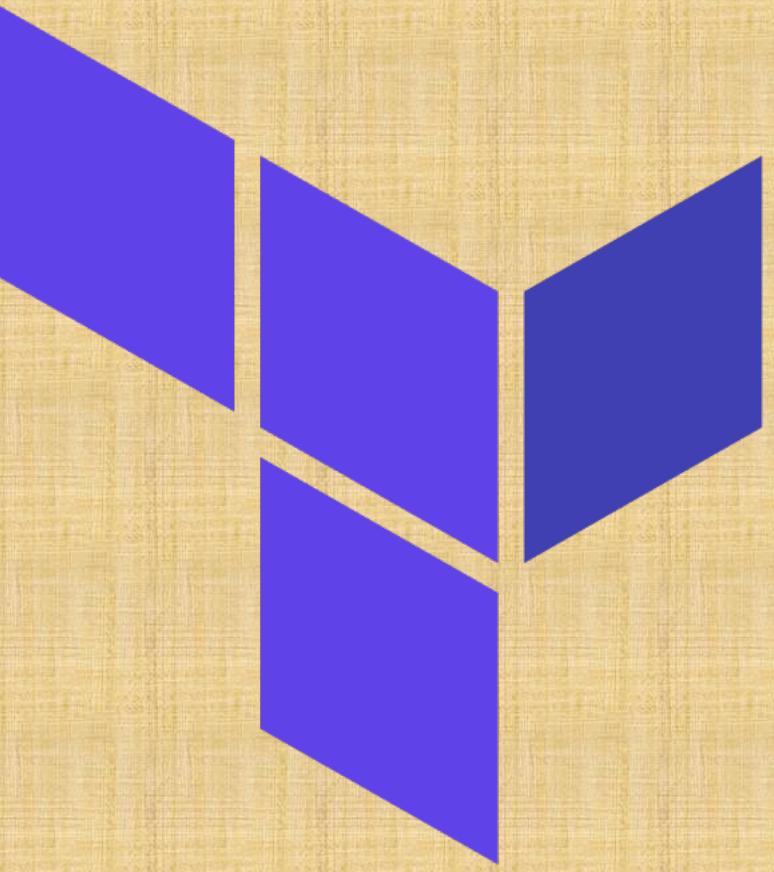
```
# Resource-8: Create EC2 Instance with Sample App
```

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 ]
}
```

Terraform Resource Meta-Argument count



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**.

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

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

Resource
Meta-Argument
count

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

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]`

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**

Use case: What are we going implement?

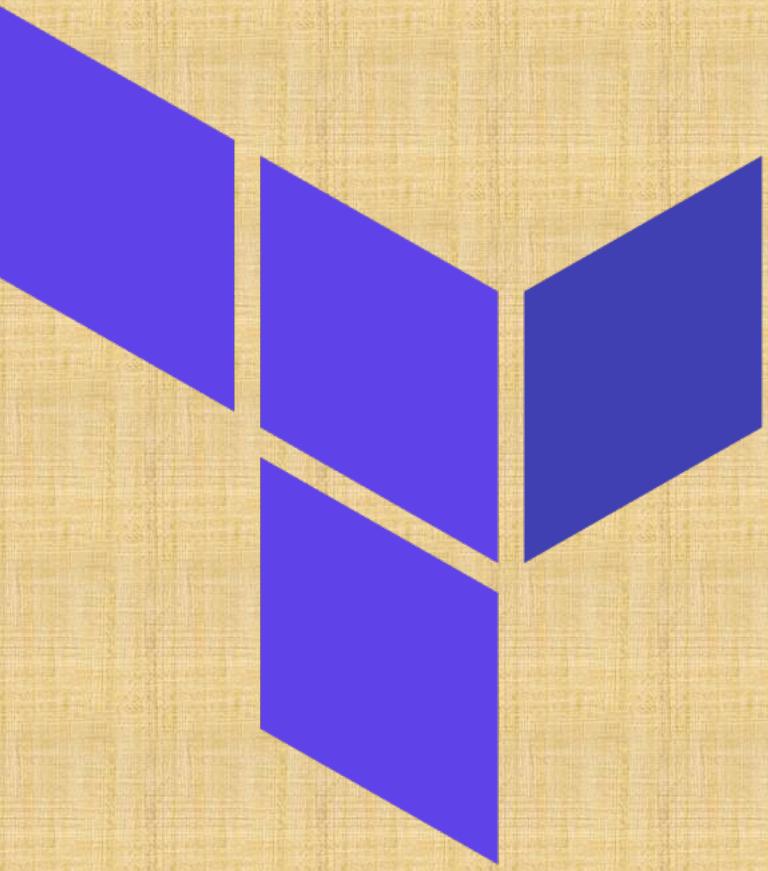
Resource-1: Use Meta-Argument Count to create multiple EC2 Instances using single Resource

The diagram illustrates the mapping of a single AWS CloudFormation resource to multiple EC2 instances. On the left, a code snippet shows the creation of a single 'aws_instance' resource with a count of 5. This resource is then mapped to five separate EC2 instances on the right, indexed from 0 to 4.

```
# 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]	

Terraform Resource Meta-Argument for_each



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.

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

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

Resource
Meta-Argument
`for_each`

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

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](#).

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

Buckets (30)	
Buckets are containers for data stored in S3. Learn more	
<input type="text"/> my	
Name	AWS Region
dev-my-dapp-bucket	US East (N. Virginia) us-east-1
prod-my-papp-bucket	US East (N. Virginia) us-east-1
qa-my-qapp-bucket	US East (N. Virginia) us-east-1
stag-my-sapp-bucket	US East (N. Virginia) us-east-1

```
# 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}"  
}
```

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

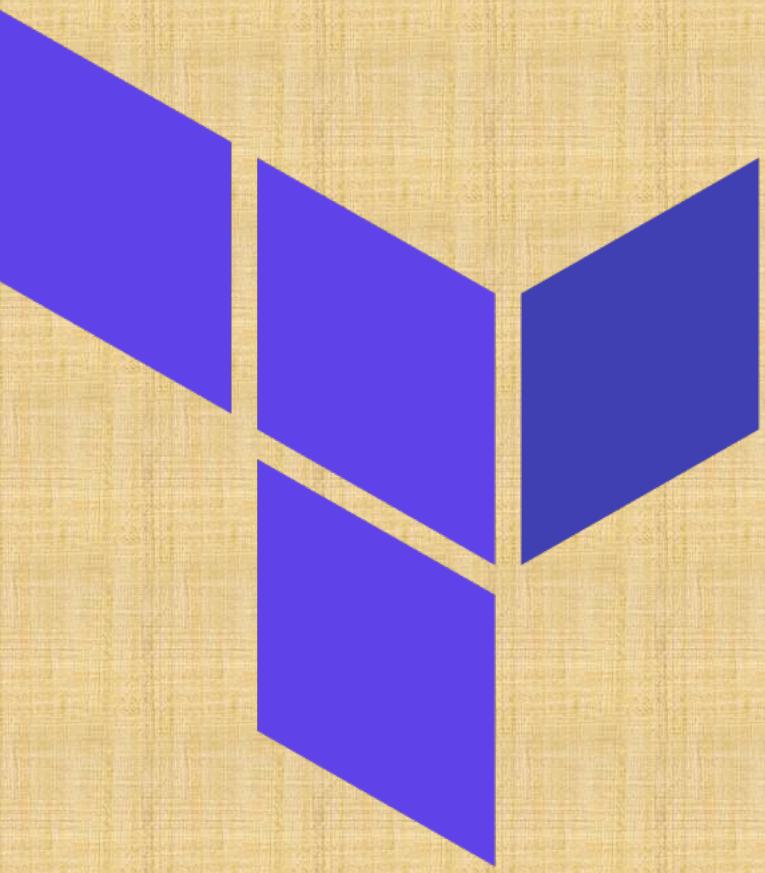
```
# Create 4 IAM Users
resource "aws_iam_user" "myuser" {
  for_each = toset( ["TJack", "TJames", "TMadhu", "TDave"] )
  name     = each.key
}
```



The screenshot shows the AWS IAM User Management console interface. At the top, there are buttons for 'Add user' (blue) and 'Delete user' (red). On the right, there are three small icons: a refresh symbol, a gear symbol, and a question mark symbol. Below the buttons is a search bar with the placeholder 'Find users by username or access key'. To the right of the search bar, it says 'Showing 7 results'. The main area is a table with the following data:

	User name	Groups	Access key age	Password age	Last activity	MFA
<input type="checkbox"/>	TMadhu	None	None	None	None	Not enabled
<input type="checkbox"/>	TJames	None	None	None	None	Not enabled
<input type="checkbox"/>	TJack	None	None	None	None	Not enabled
<input type="checkbox"/>	TDave	None	None	None	None	Not enabled

Terraform Resource Meta-Argument lifecycle



Resource Meta-Argument lifecycle

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

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

create_before_destroy

prevent_destroy

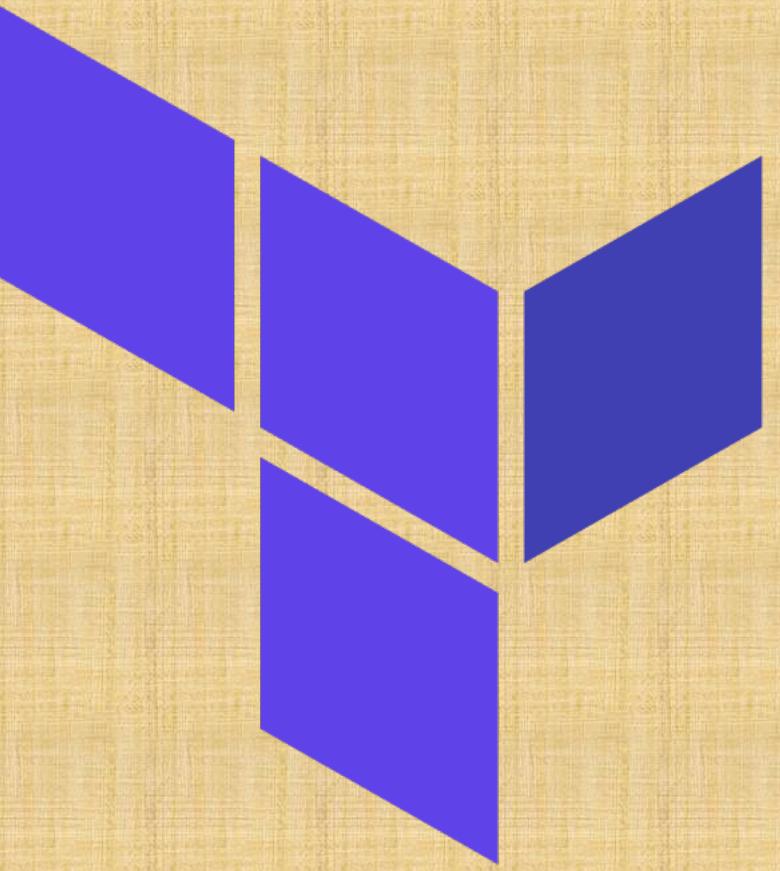
ignore_changes

```
# 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
  }
}
```

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

```
# 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,
    ]
  }
}
```

Terraform Variables Introduction



Terraform Variables

Terraform
Input
Variables

Terraform
Output
Values

Terraform
Local
Values

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.

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

Terraform
Input
Variables

Terraform Variables – Output Values

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

1

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

Terraform
Variables
Outputs

2

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

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

3

Advanced

DRY Principle

Don't Repeat Yourself

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  
}
```

Practical Examples with Step-by-Step Documentation on Github

master ▾ hashicorp-certified-terraform-a

stacksimplify Welcome to Stack Simplify

..

- 05-01-Terraform-Input-Variables
- 05-02-Terraform-Output-Values
- 05-03-Terraform-Local-Values

05-Terraform-Variables
05-01-Terraform-Input-Variables
v1-Input-Variables-Basic
v2-Input-Variables-Assign-when-prompted
v3-Input-Variables-Override-default-with-cli-var
v4-Input-Variables-Override-with-Environment-Variables
v5-Input-Variables-Assign-with-terraform-tfvars
v6-Input-Variables-Assign-with-tfvars-var-file
v7-Input-Variables-Assign-with-auto-tfvars
v8-01-Input-Variables-Lists
v8-02-Input-Variables-Maps
v9-Input-Variables-Validation-Rules
v10-Sensitive-Input-Variables
v11-File-Function
README.md
05-02-Terraform-Output-Values
terraformer-manifests
apache-install.sh
c1-versions.tf
c2-variables.tf
c3-security-groups.tf
c4-ec2-instance.tf
c5-outputs.tf
README.md
05-03-Terraform-Local-Values
terraformer-manifests
c1-versions.tf
c2-variables.tf
c3-s3-bucket.tf
README.md

Terraform Variables - Duration

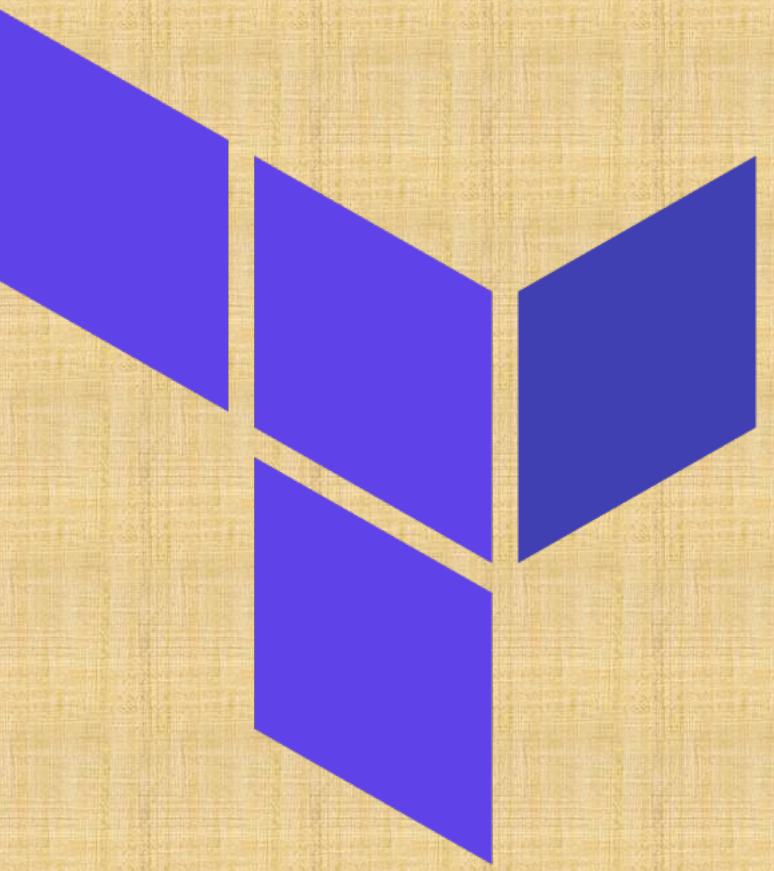
^ Terraform Variables

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

Terraform Variables

Input Variables



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.

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

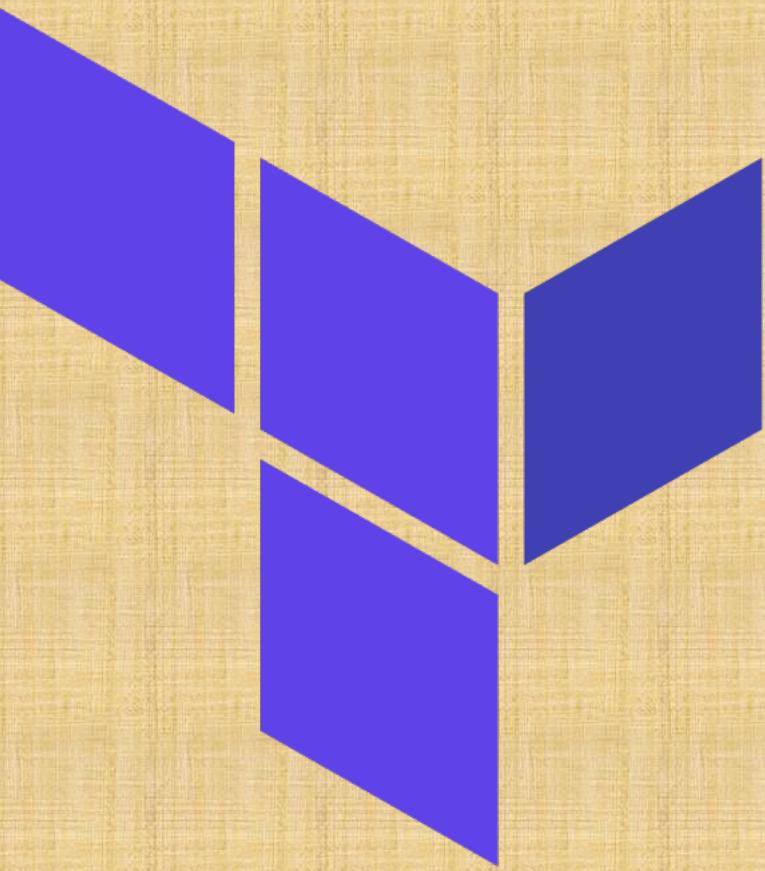
8 Implement complex type **constructors** like `List` & `Map` in Input Variables

9 Implement **Custom Validation Rules** in Variables

10 Protect **Sensitive** Input Variables

Terraform
Input
Variables

Terraform Variables Output Values



Terraform Variables – Output Values

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

1

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

Terraform
Variables
Outputs

2

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

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

3

Advanced

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
}
```

Terraform Variables - Output Values

```
aws_instance.my-ec2-vm: Creation complete after 24s [id=i-0406c673]
```

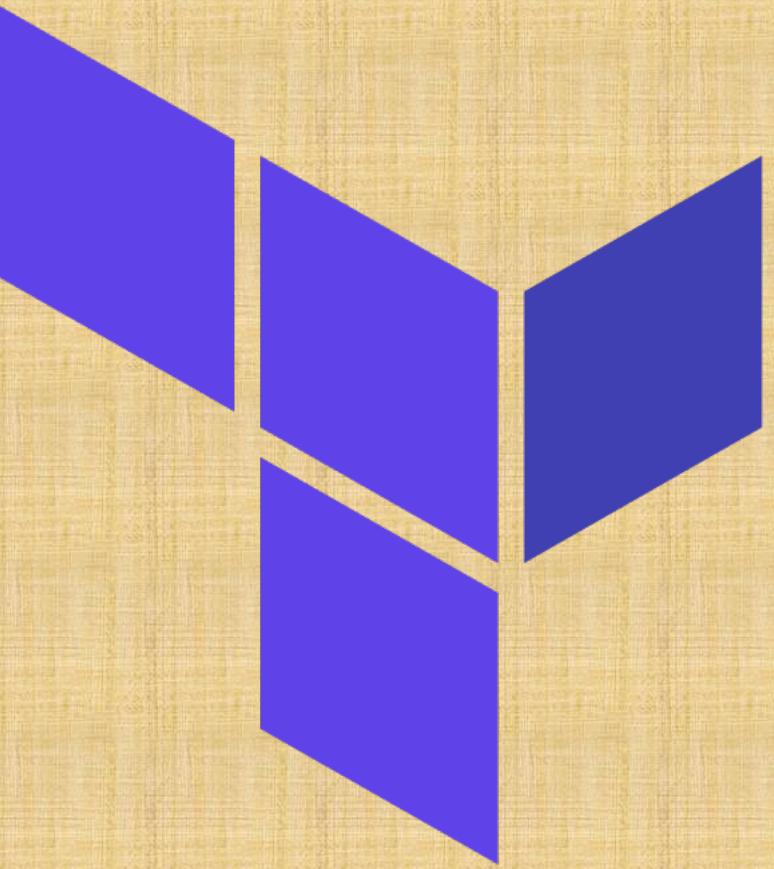
```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.
```

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",
])
```

Kubernetes Cluster Configuration

Terraform Variables Local Values



DRY Principle

Don't Repeat Yourself

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  
    }  
}
```

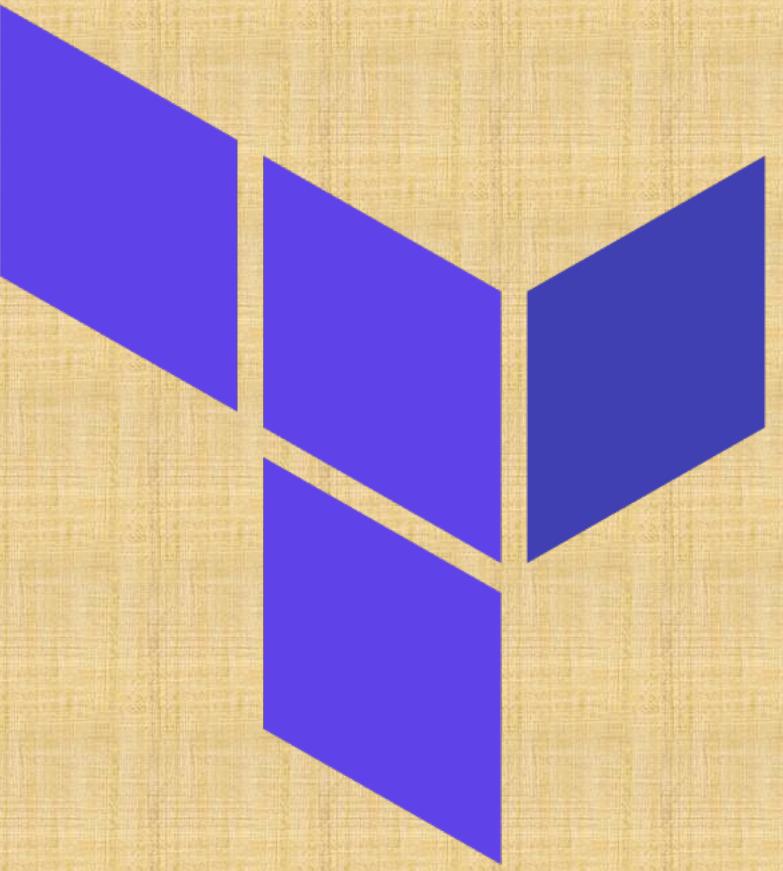
```
resource "aws_instance" "example" {  
    # ...  
  
    tags = local.common_tags  
}
```

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

Terraform Datasources



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"]
    }
}
```

Terraform Datasources

We can refer the data resource in a resource as depicted

Meta-Arguments for Datasources

```
# 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"
    }
}
```

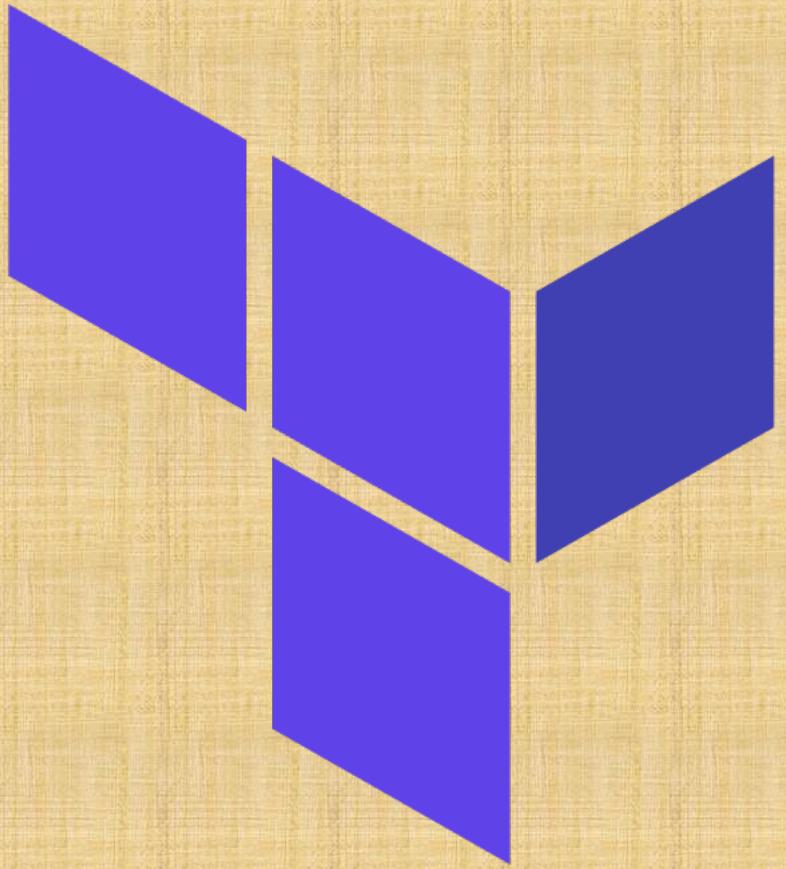
Data resources support the **provider** meta-argument as defined for managed resources, with the **same syntax** and behavior.

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**.

Terraform State Introduction



Terraform State

Terraform
Remote
State
Storage

Terraform
Commands
from
State
Perspective

What is Terraform Backend ?

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

Terraform
State Storage



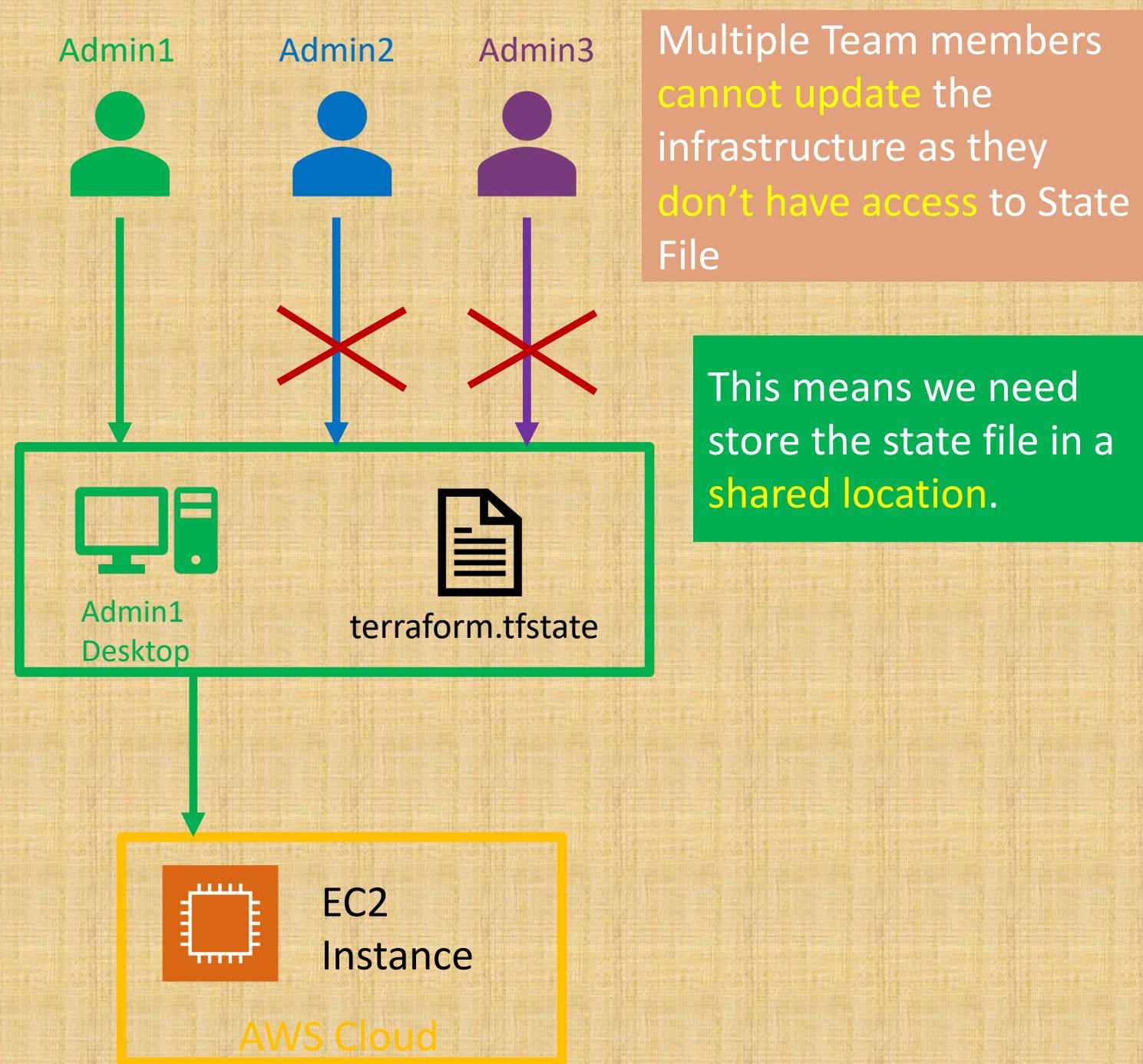
AWS S3 Bucket

Terraform
State Locking

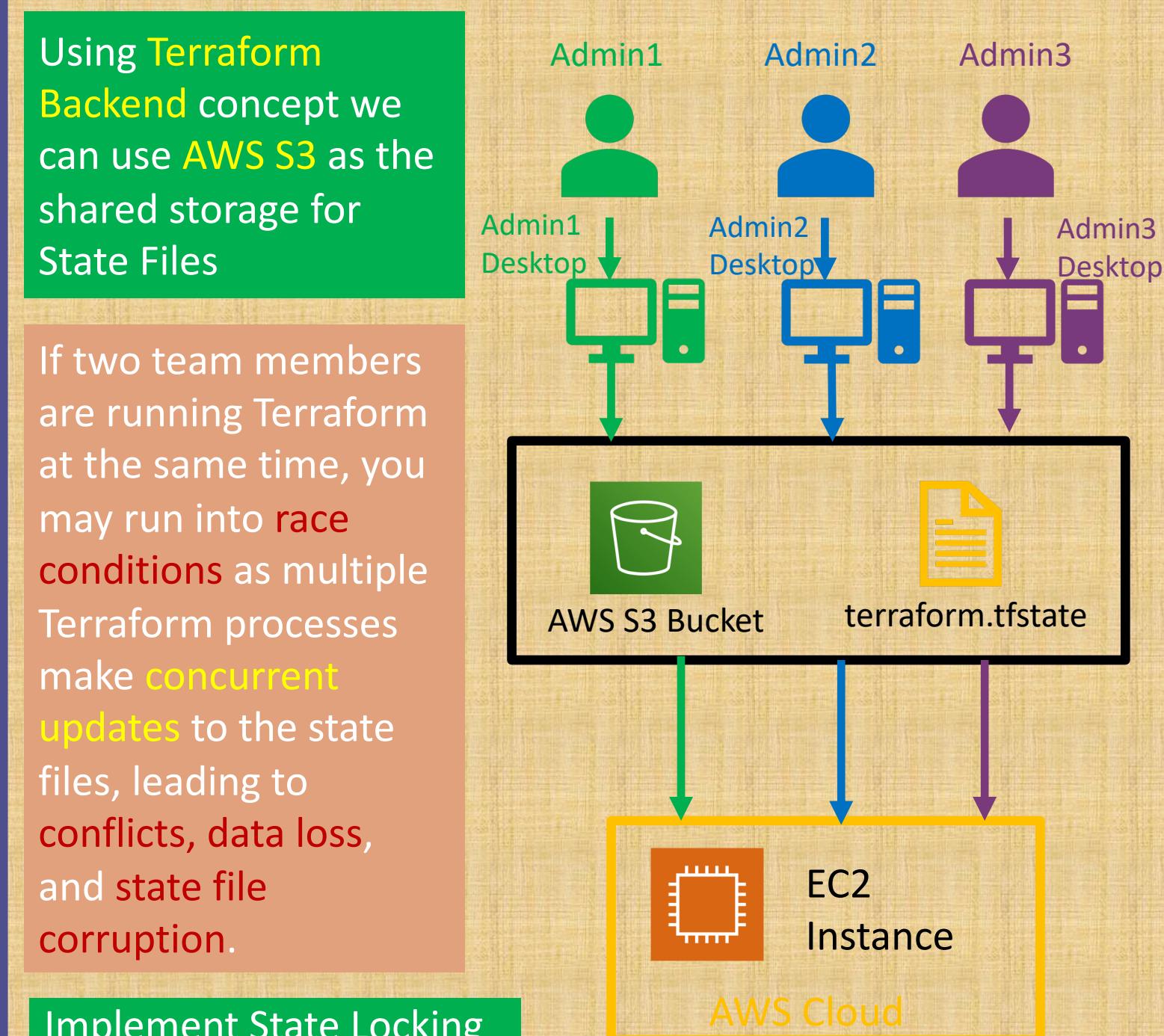


AWS DynamoDB

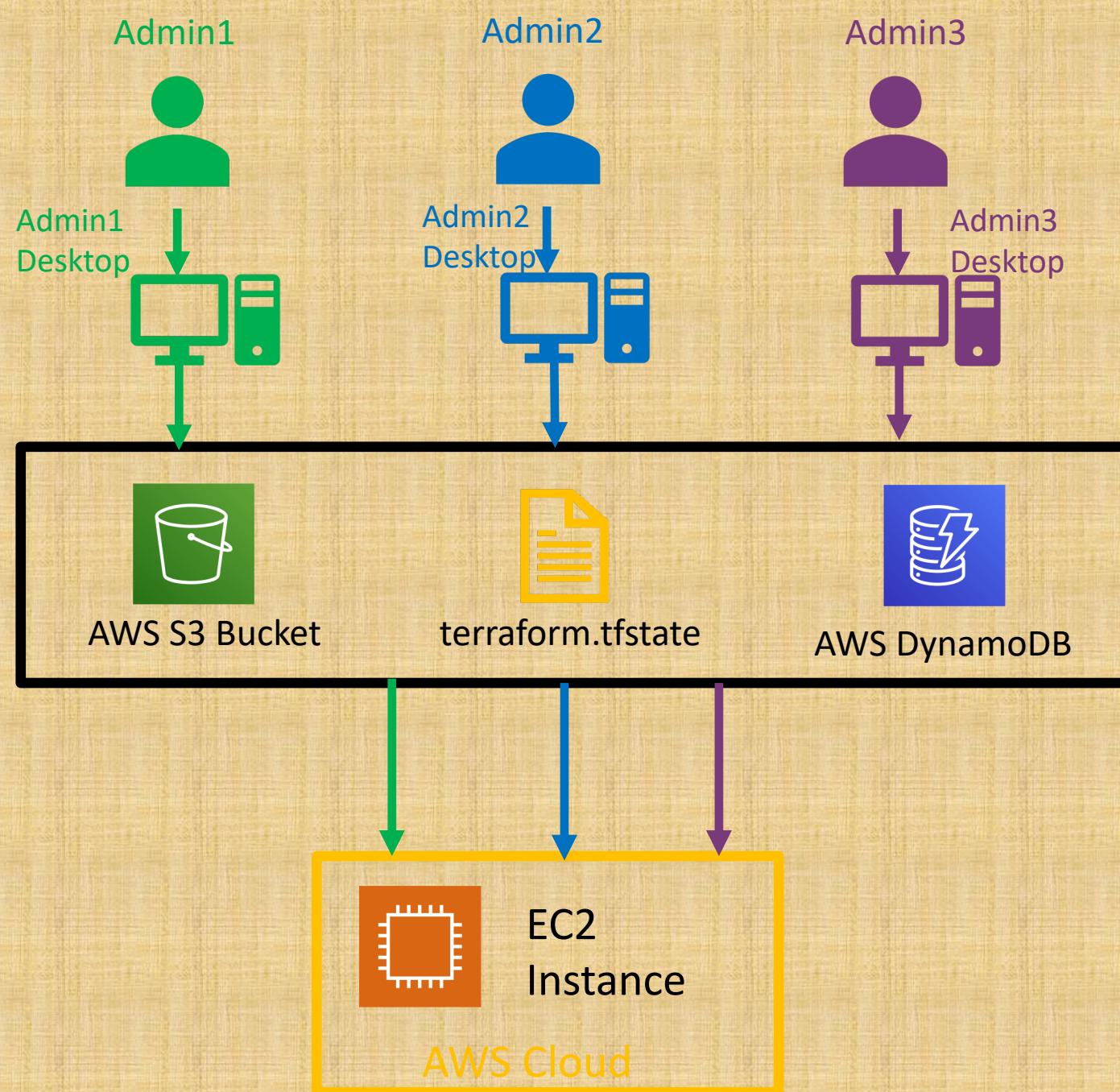
Local State File



Remote State File



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.

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"
```

Terraform Commands – State Perspective

terraform show

terraform refresh

terraform plan

terraform state

Terraform
Commands

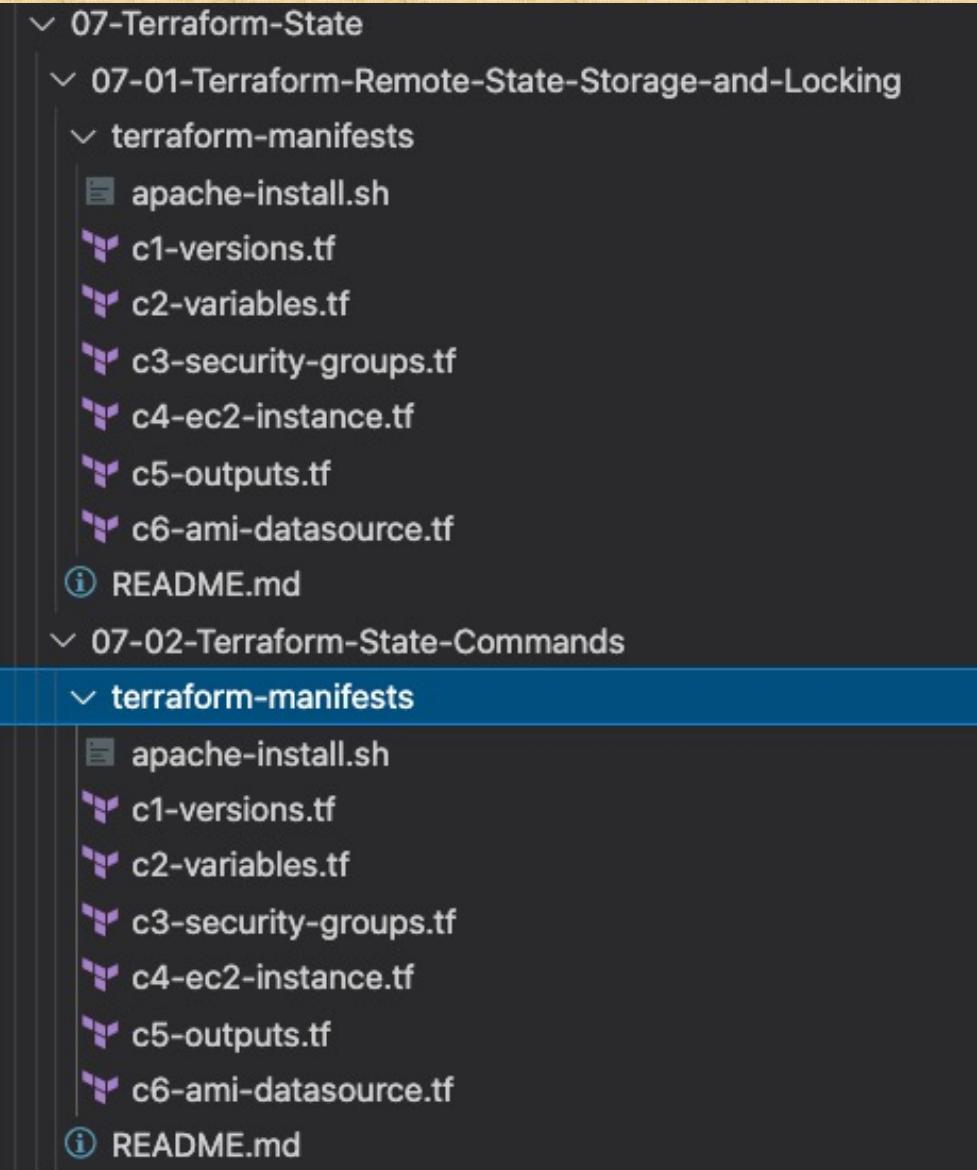
terraform
force-unlock

terraform taint

terraform untaint

terraform
apply target

Practical Examples with Step-by-Step Documentation



The screenshot shows a video player interface for a course titled "hashicorp-certified-terraform-associate / Terraform State".

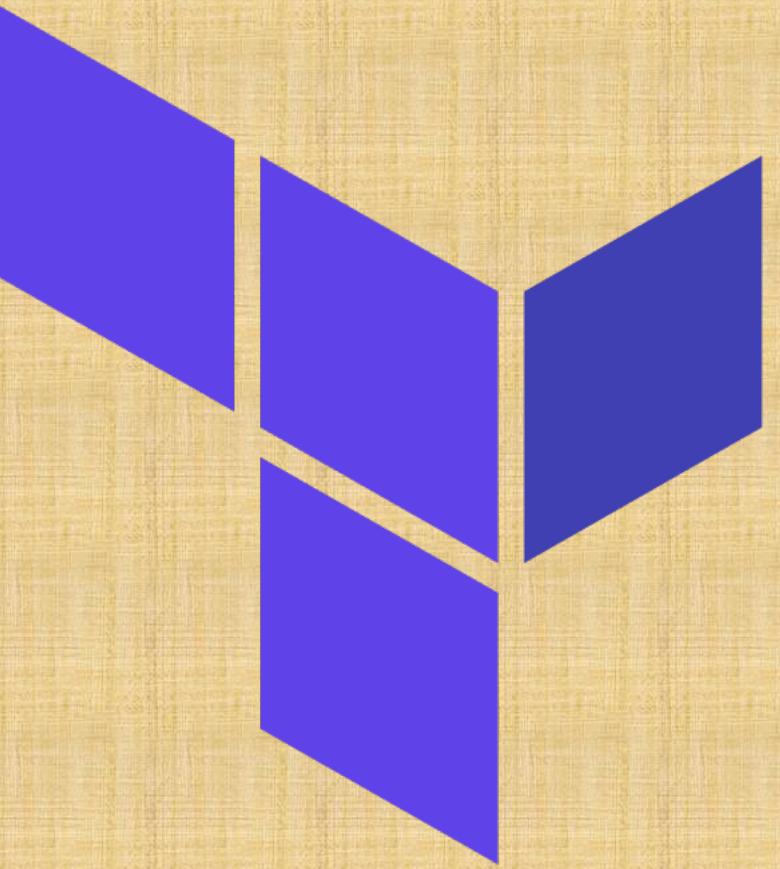
Course navigation:

- stacksimplify Welcome to Stack Simplify
- ..
- 07-01-Terraform-Remote-State-Storage-and-... Welcome to
- 07-02-Terraform-State-Commands Welcome to

Course details:

- Terraform State** (12 lectures • 1hr 37min)
- Step-01: Remote State Storage Introduction (10:02)
- Step-02: Configure AWS S3 as Terraform Backend for Remote State Storage (10:45)
- Step-03: Implement State Locking using AWS DynamoDB (06:06)

Terraform Backend Remote State Storage



What is Terraform Backend ?

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

Terraform
State Storage



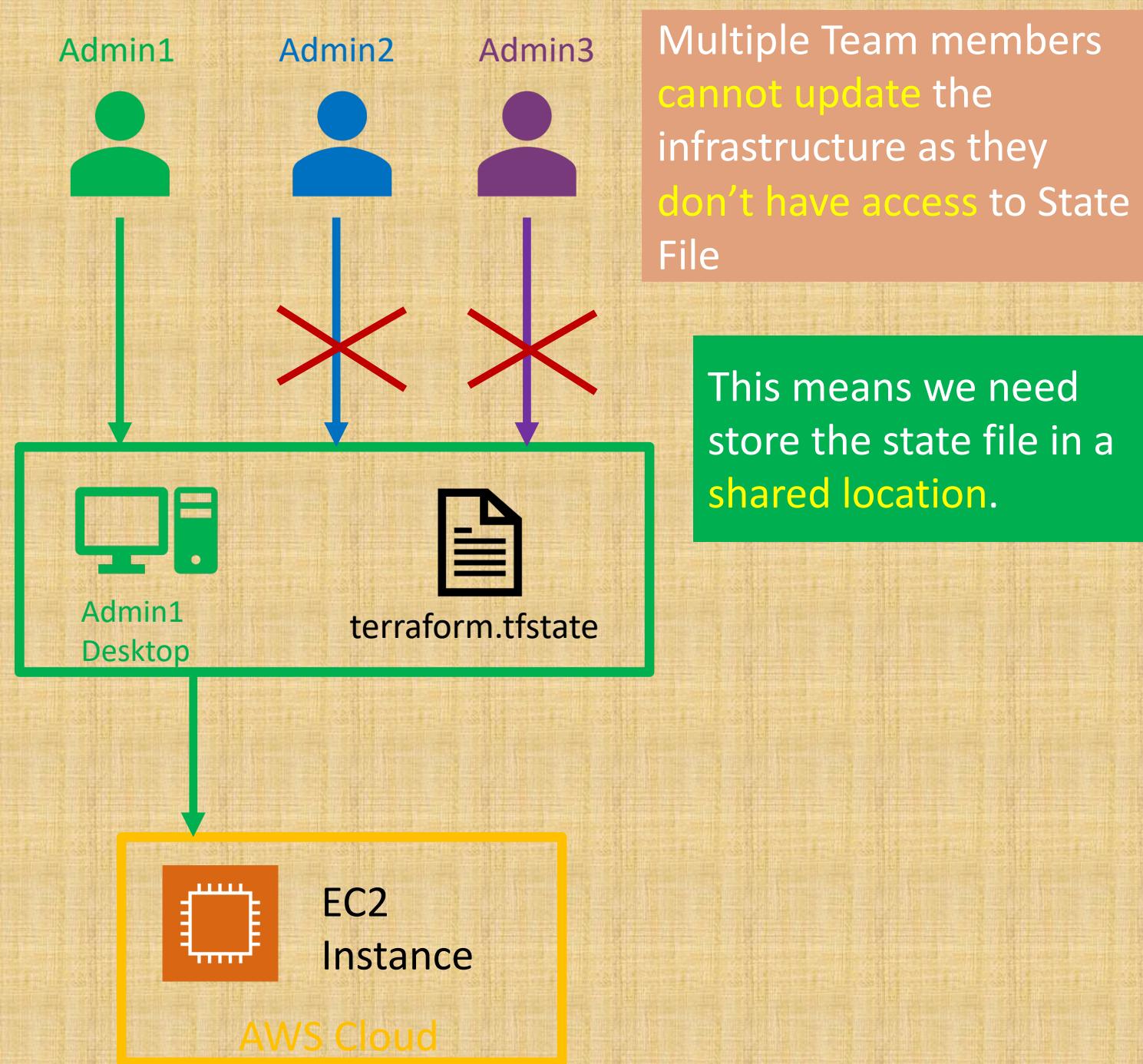
AWS S3 Bucket

Terraform
State Locking

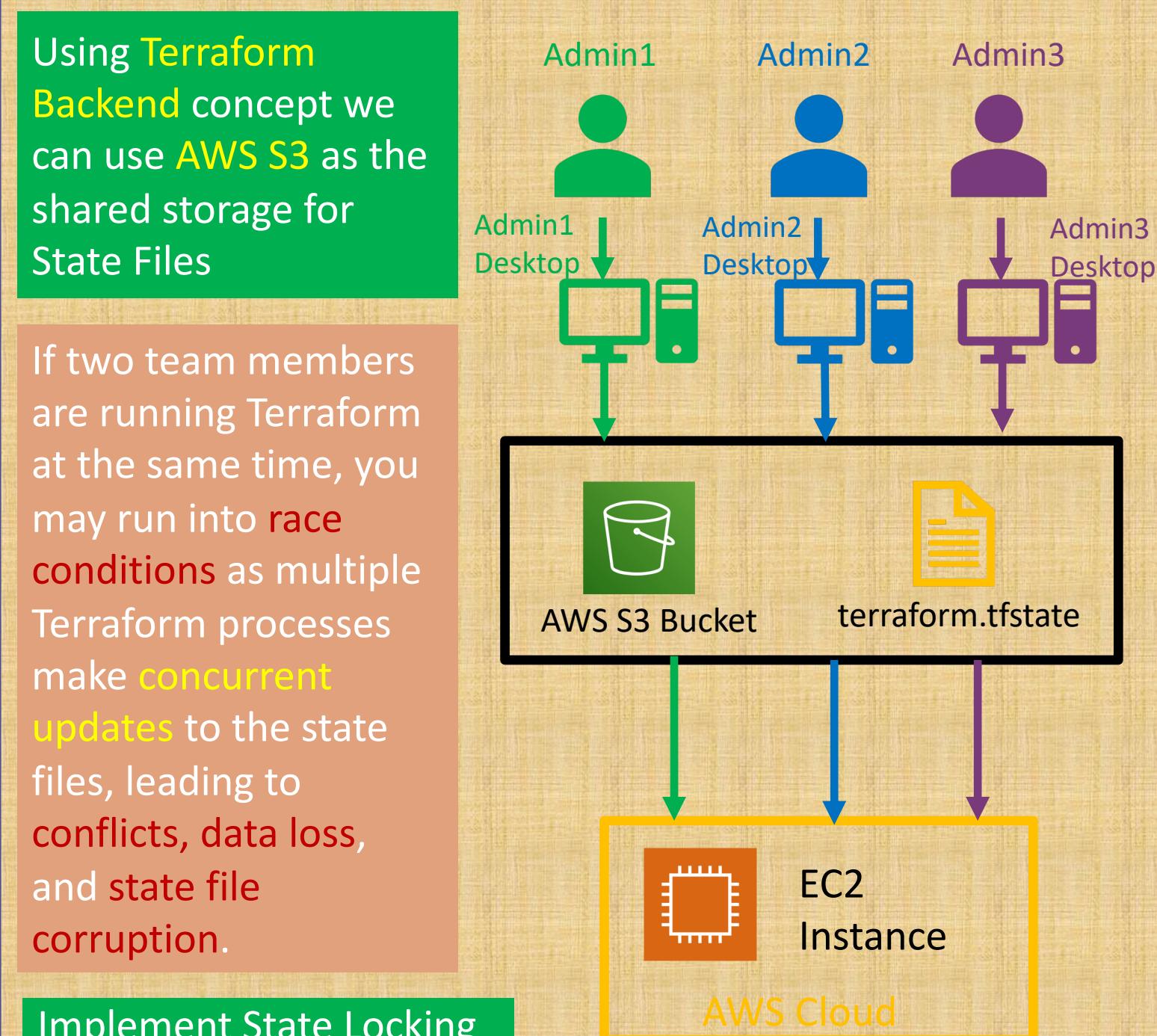


AWS DynamoDB

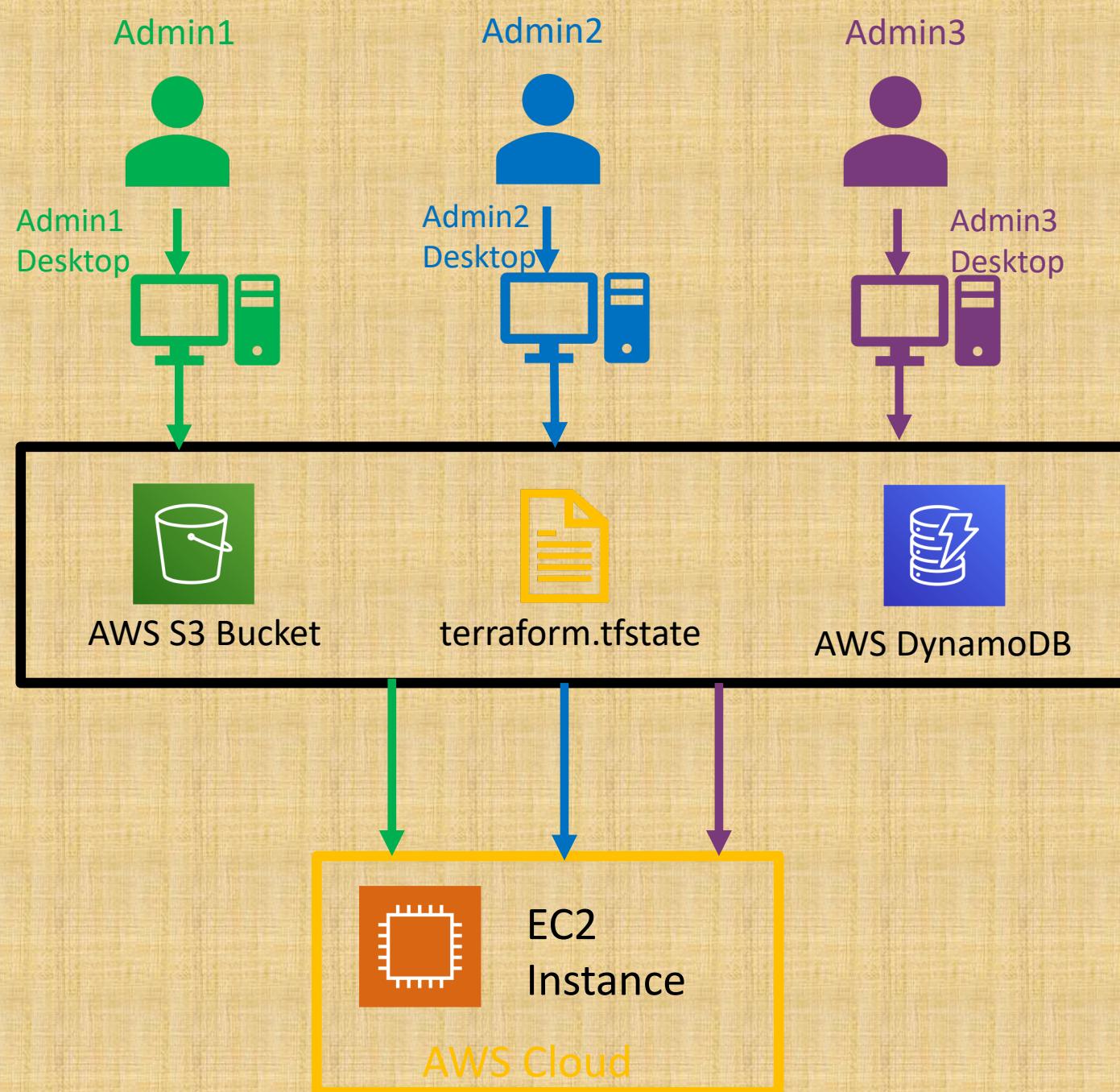
Local State File



Remote State File



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.

Terraform Remote State File with State Locking

Terraform State Storage to Remote Backend

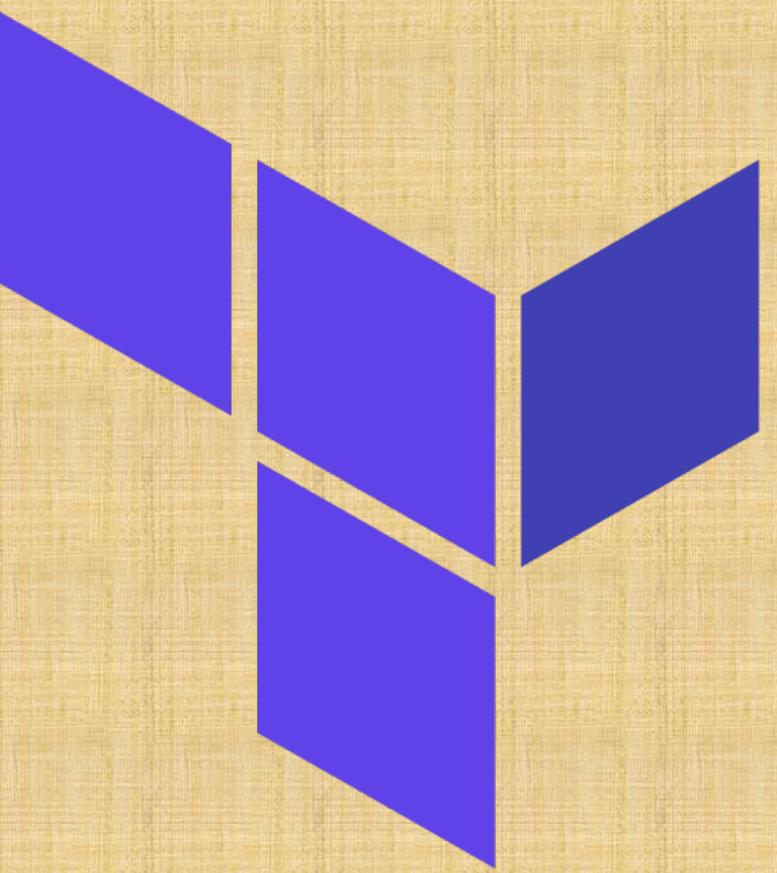
Terraform State Locking

```
# 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"
```

Terraform Backends



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

Terraform Backends

What Backends Do

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

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.

What are Operations ?
terraform apply
terraform destroy

Terraform Backends

Backend Types

Enhanced Backends

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

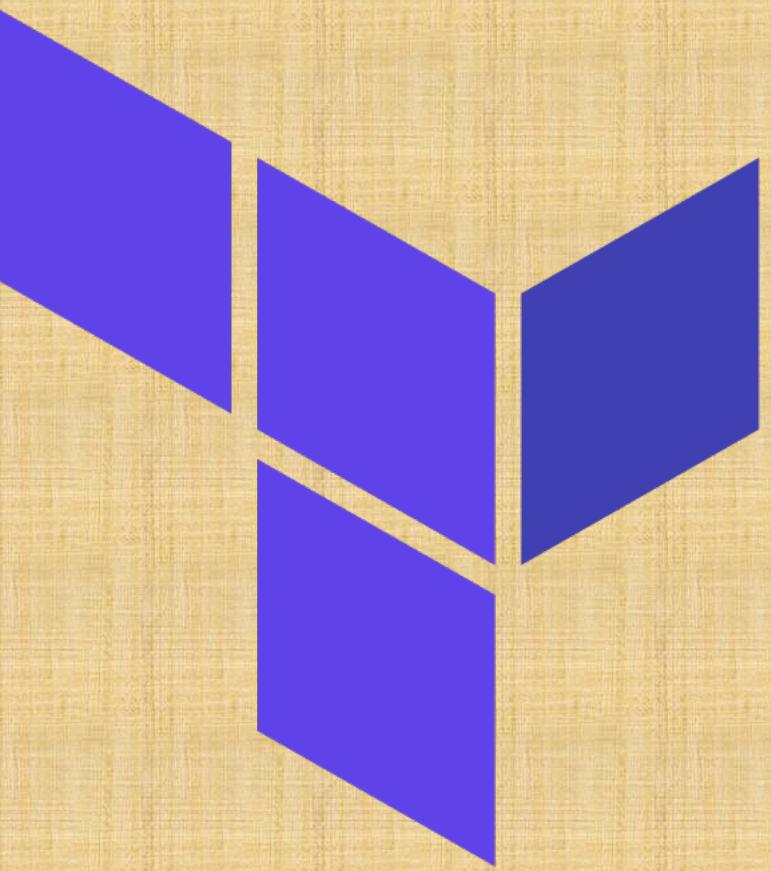
Example for Remote Backend
Performing Operations : Terraform Cloud, Terraform Enterprise

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

Terraform Workspaces



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

Terraform Workspace Commands

Terraform Workspace
Commands

`terraform workspace show`

`terraform workspace list`

`terraform workspace new`

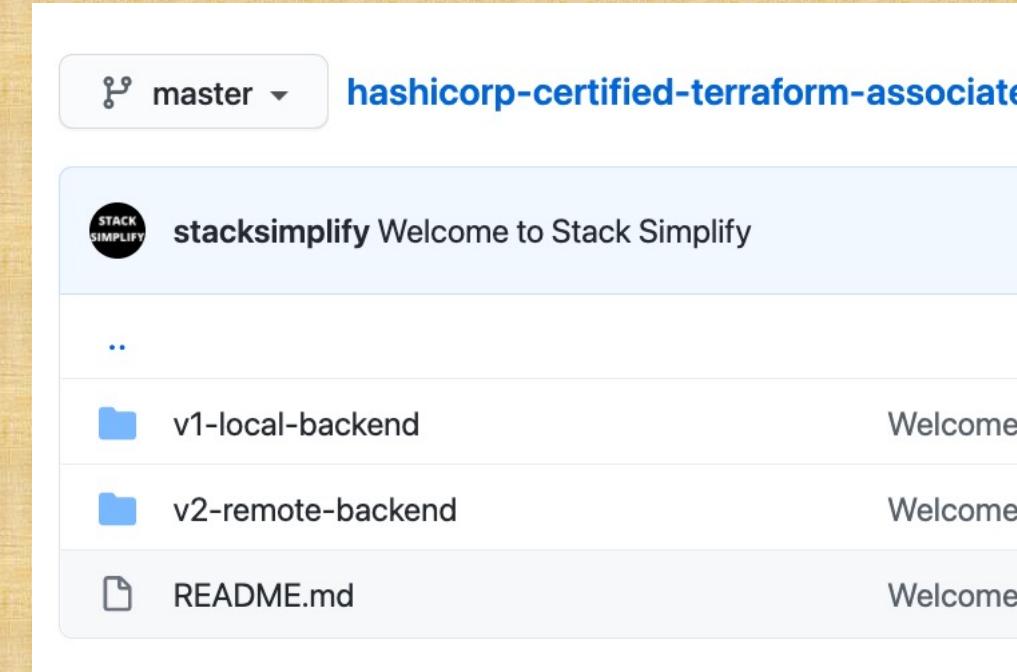
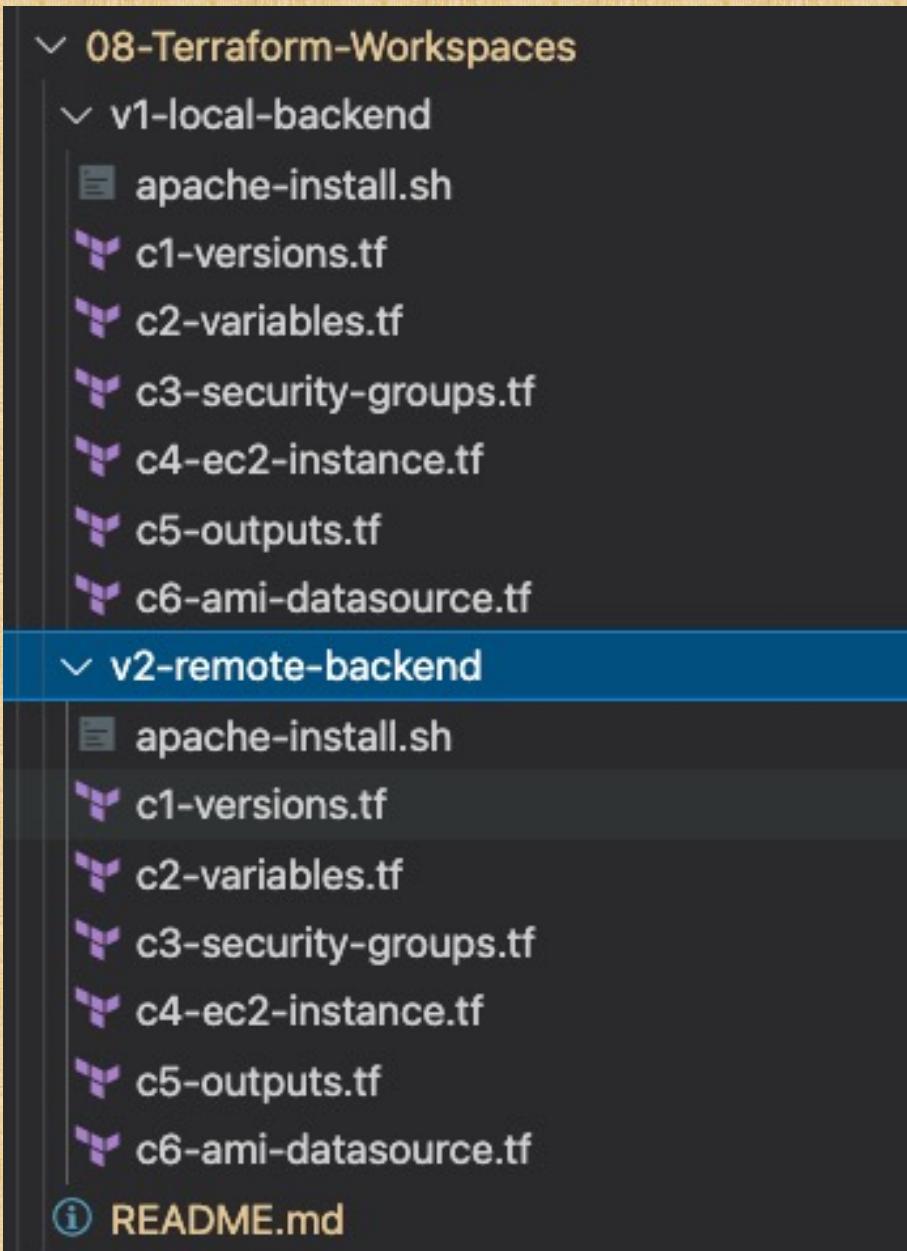
`terraform workspace select`

`terraform workspace delete`

Usecase-1: Local Backend

Usecase-2: Remote Backend

Practical Example with Step-by-Step Documentation on Github



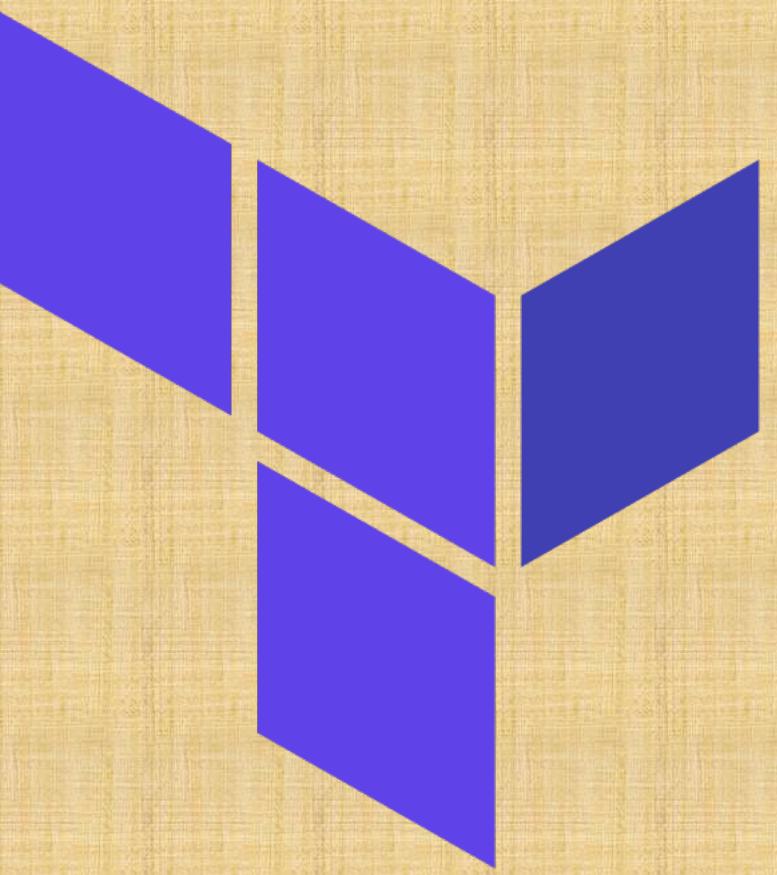
The screenshot shows a video player interface for a course section titled 'Terraform Workspaces'. The video details are as follows:

- 5 lectures • 40min (highlighted with a green border)
- 07:53
- 07:30

The video content includes:

- Step-02: Review / Create Terraform Manifests to support multiple workspaces
- Step-03: Local Backend: Create Resources in default

Terraform Provisioners



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.

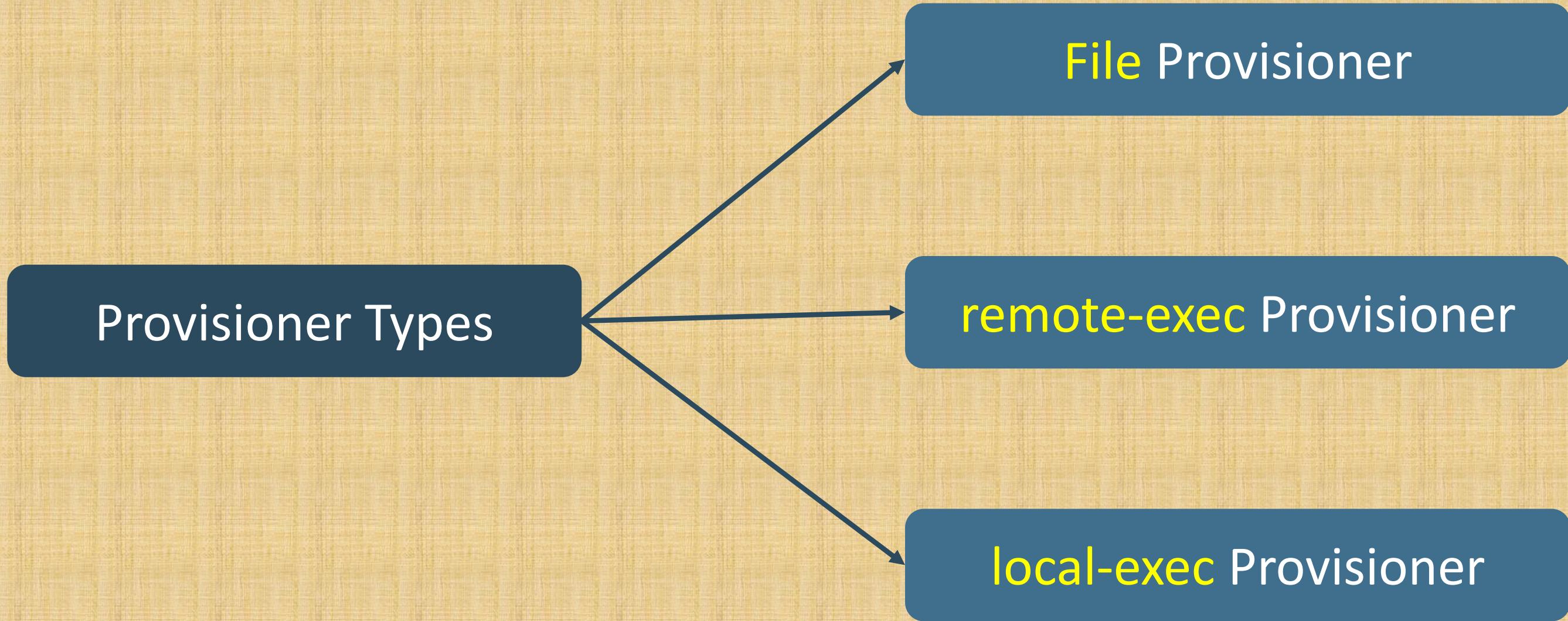
Provisioners are a Last Resort

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

Types of Provisioners



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")
}
```

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 Provisioner
# Connection Block for Provisioners to connect to EC2 Instance
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 – FOLDED
provisioner "file" {
  source      = "apps/app1"
  destination = "/tmp"
}
```

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/"
}
```

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"
  ]
}
```

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"
}
```

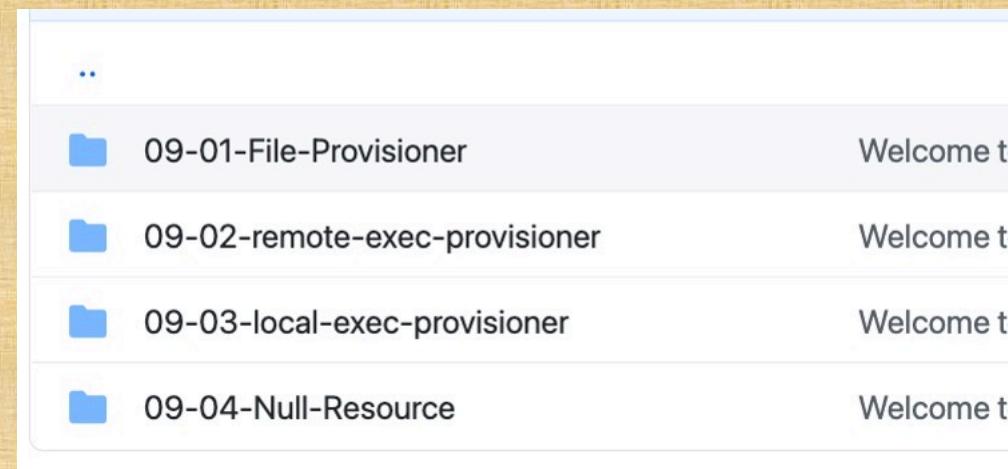
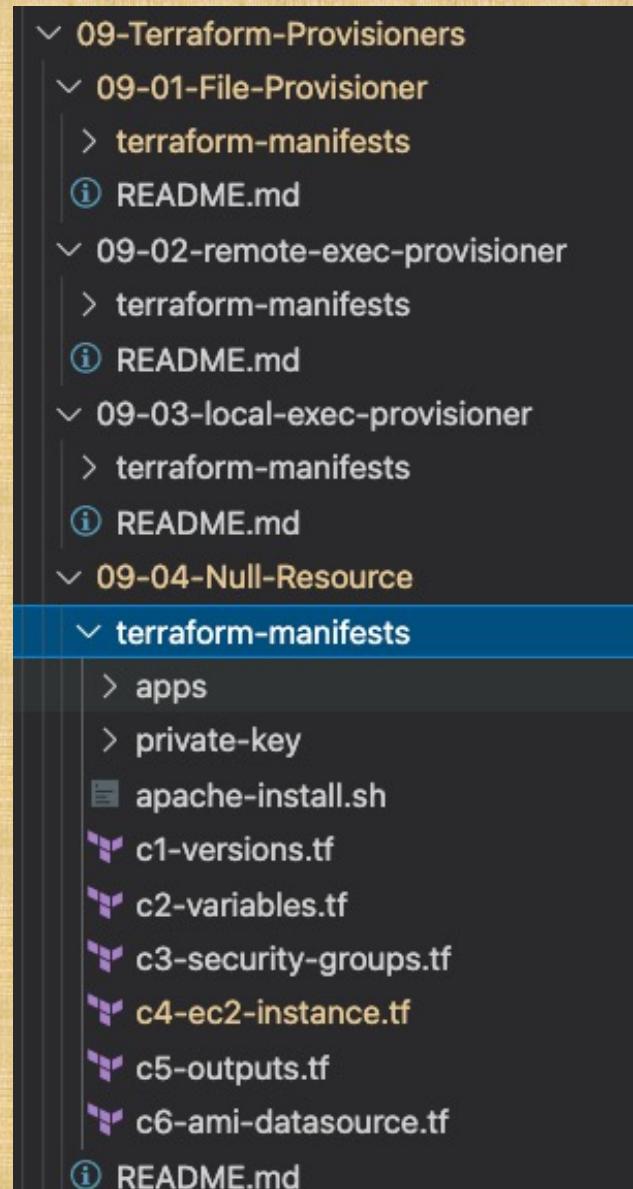
```
# Sync App1 Static Content to Webserver using P
resource "null_resource" "sync_app1_static" {
  depends_on = [ time_sleep.wait_90_seconds ]
  triggers = {
    always-update = timestamp()
  }
}
```

```
# 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/www/html
provisioner "remote-exec" {
  inline = [
    "sudo cp -r /tmp/app1 /var/www/html"
  ]
}
```

Practical Examples & Step-by-Step Documentation on Github

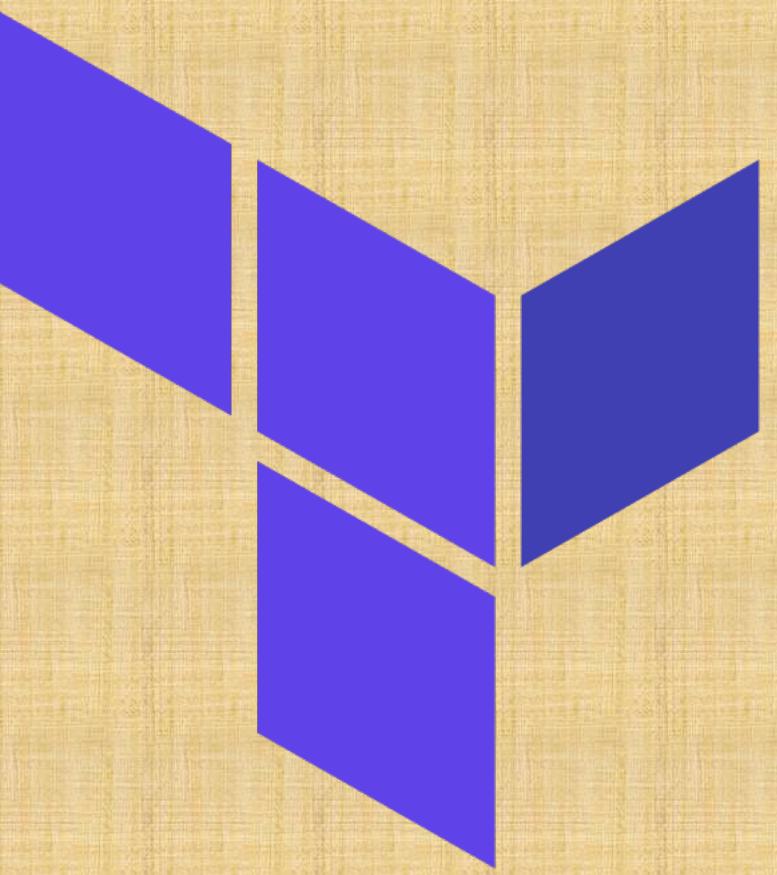


^ **Terraform Provisioners & Null Resource**

7 lectures • 1hr 8min

- Step-01: Introduction to Terraform Provisioners
- Step-02: Implement Connection Block for File Provisioners & Review Manifests 12:10
- Step-03: File Provisioner: Execute Terraform Commands to Verify 10:44

Terraform Modules



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.

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**.

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

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

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

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.

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.

Modules - Demo-1

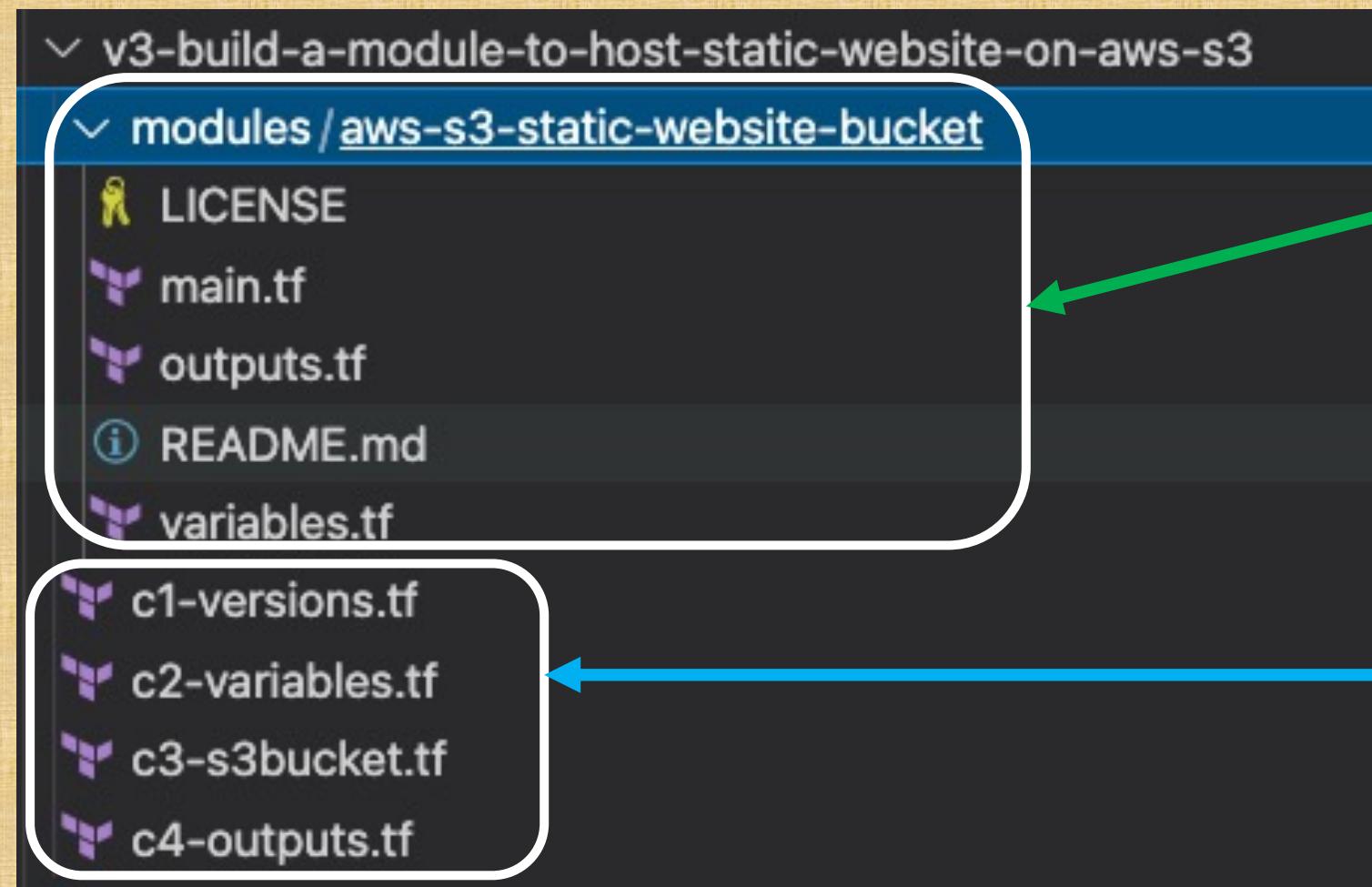
```
# AWS EC2 Instance Module
module "ec2_cluster" {
    source          = "terraform-aws-modules/ec2-instance/aws"
    version         = "~> 2.0"

    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 Security Group ID
    subnet_id        = "subnet-4ee95470" # Get one public subnet
    user_data        = file("apache-install.sh")

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

Modules - Demo-2



Child Module

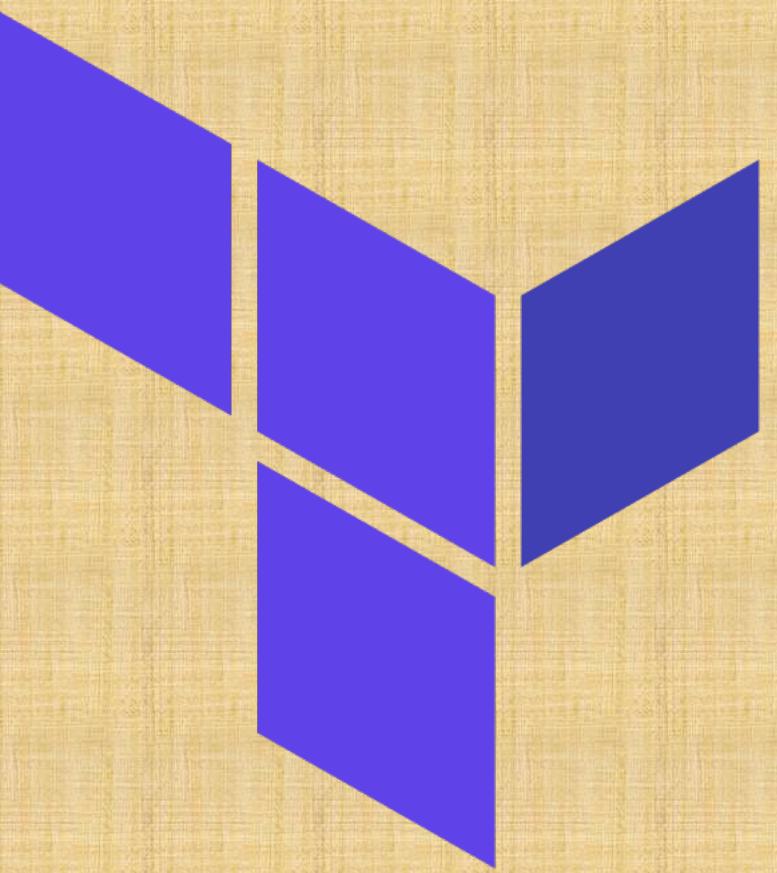
Root Module

Practical Examples & Step-by-Step Documentation on Github

A screenshot of a GitHub repository interface. The repository name is partially visible as 'hashicorp-certified-terraform-associate /'. The sidebar shows a file tree for '10-Terraform-Modules' under '10-01-Terraform-Modules-Basics'. The 'terraform-manifests' folder is expanded, showing files like 'apache-install.sh', 'c1-versions.tf', 'c2-variables.tf', 'c3-ami-datasource.tf', 'c4-ec2instance-module.tf', 'c5-outputs.tf', and 'README.md'. Other collapsed sections include '10-02-Terraform-Build-a-Module' and several 'vX' sections.

A screenshot of a StackSimplify course page titled 'Terraform Modules'. The course summary indicates '10 lectures • 1hr 27min'. Two lessons are listed: 'Step-02: Defining Child Modules - Create EC2 Instances using Terraform Modules' (duration 10:53) and 'Step-03: Create Outputs for EC2 Instance Module, Execute Commands and Test' (duration 11:38). A green box highlights the course duration.

Terraform Cloud



Terraform Cloud Or Terraform Enterprise

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

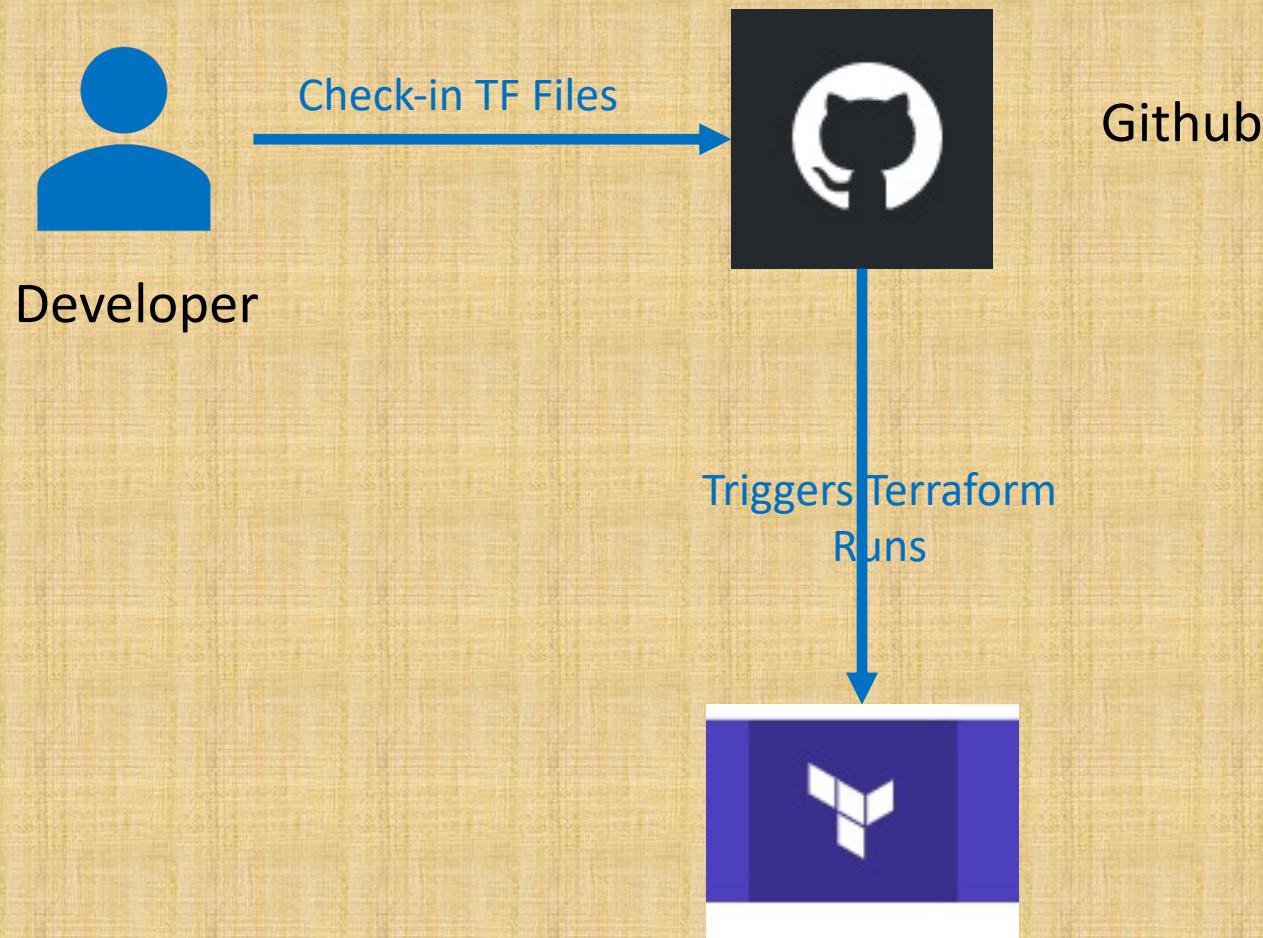
It manages **Terraform runs** in a **consistent and reliable environment**, and includes easy access to **shared state** and **secret data**, **access controls** for approving changes to infrastructure, a **private registry** for sharing Terraform modules, **detailed policy controls** for governing the contents of Terraform configurations,

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

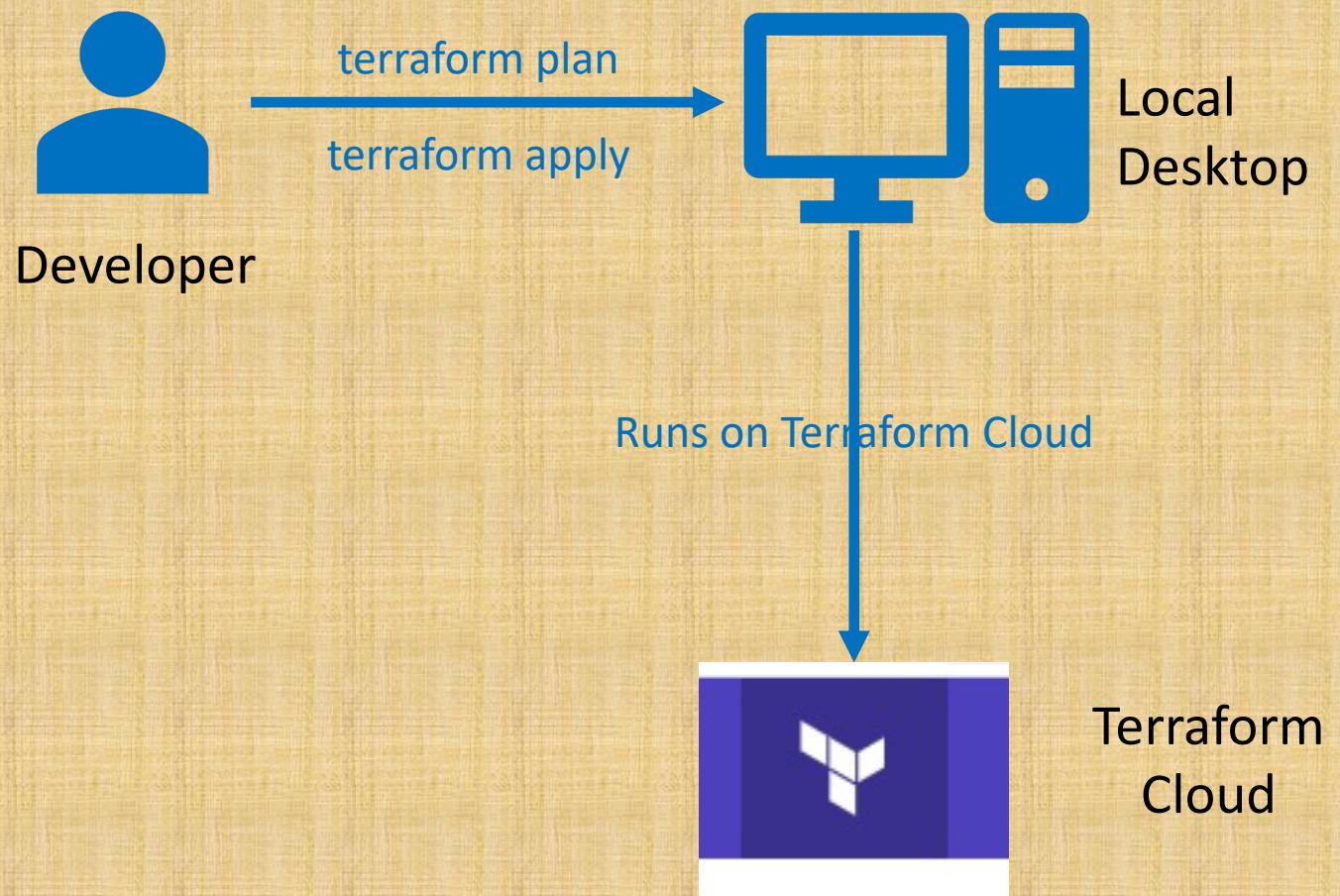
They offer **free accounts** for small teams, and **paid plans** with additional feature sets for medium-sized businesses.

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

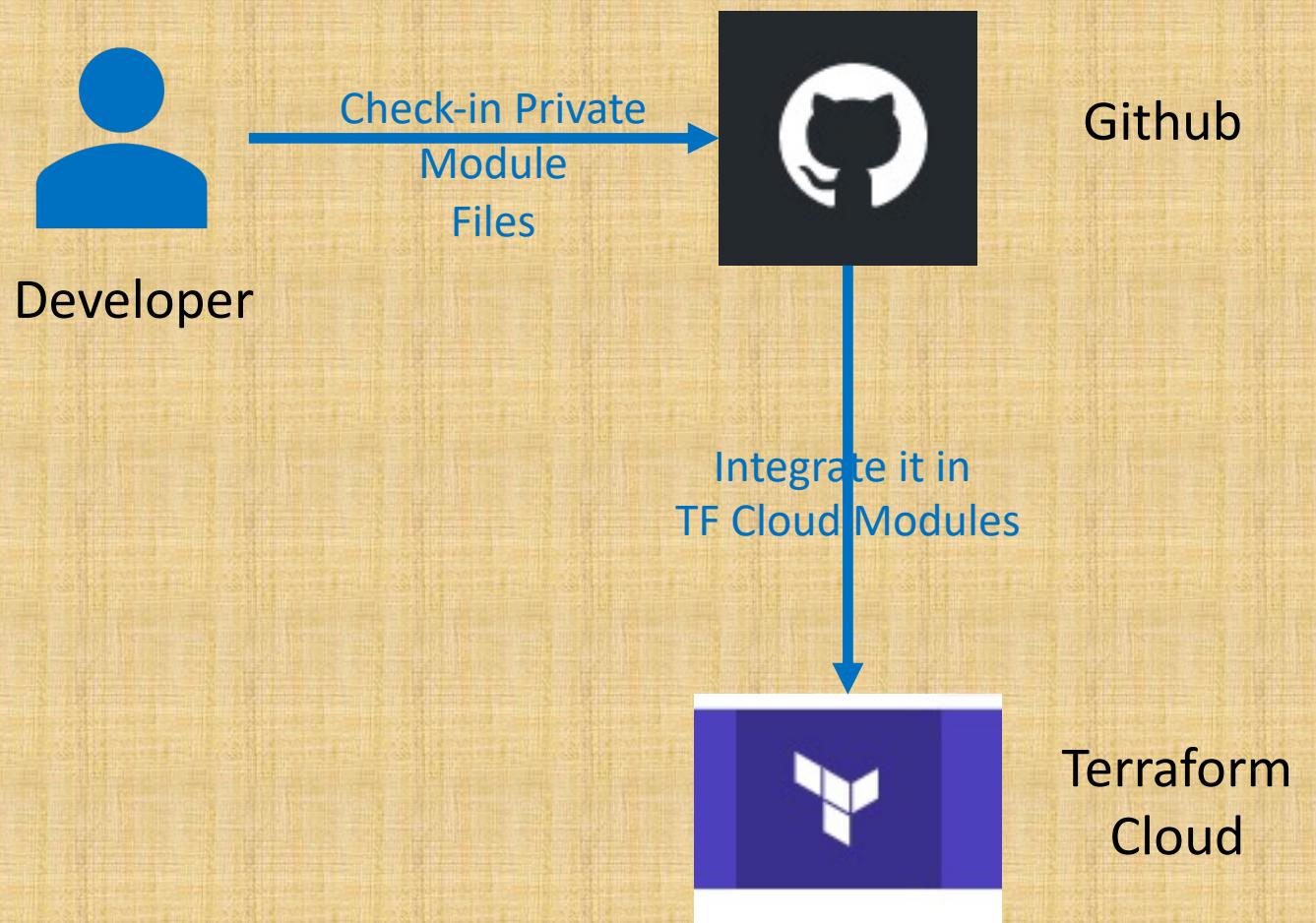
Terraform VCS-Driven Workflow



Terraform CLI-Driven Workflow



Publish Private Module Registry in Terraform Cloud



Practical Examples with Step-by-Step Github Documentation

- ✓ 11-Terraform-Cloud-and-Enterprise-Capabilities
 - ✓ 11-01-Terraform-Cloud-Github-Integration
 - > [terraform-manifests](#)
 - [\(i\) README.md](#)
 - ✓ 11-02-Share-Modules-in-Private-Module-Registry
 - > [static-file](#)
 - > [terraform-manifests](#)
 - > [terraform-s3-website-module-manifests](#)
 - [\(i\) README.md](#)
 - ✓ 11-03-Terraform-Cloud-CLI-Driven-Workflow
 - > [static-file](#)
 - > [terraform-manifests](#)
 - [\(i\) README.md](#)
 - ✓ 11-04-Migrate-State-to-Terraform-Cloud
 - > [terraform-manifests](#)
 - [\(i\) README.md](#)

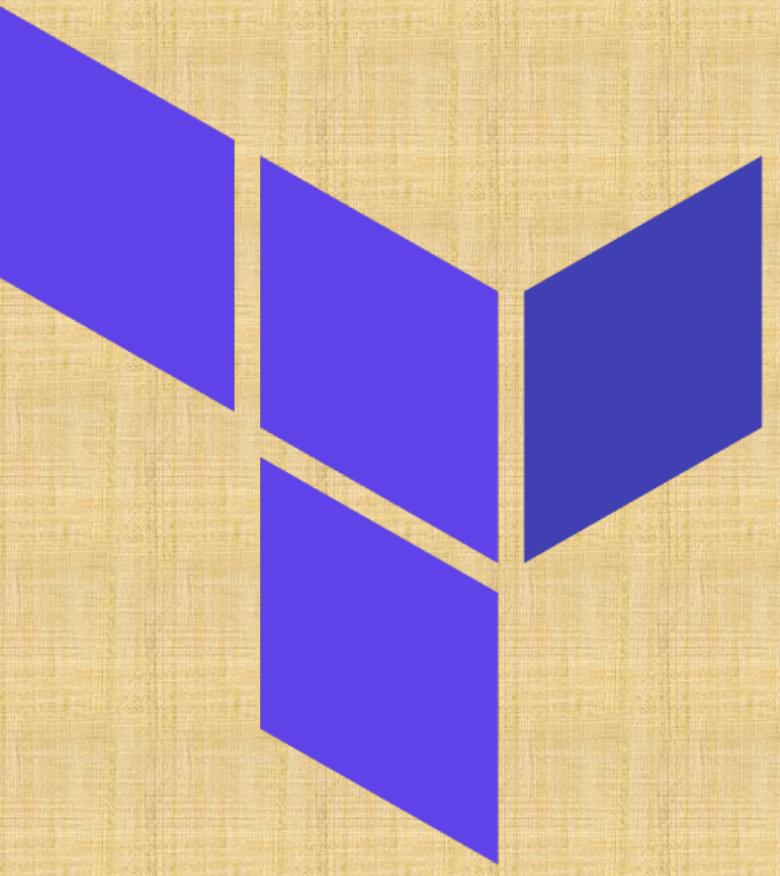
The screenshot shows a video player interface with a list of 13 lectures under the title "Terraform Cloud & Enterprise Capabilities". The first four lectures are visible in the main list, while the rest are shown in a scrollable area below. A green box highlights the duration of the first two lectures.

Lecture	Title	Duration
1	11-01-Terraform-Cloud-Github-Integration	10:49
2	11-02-Share-Modules-in-Private-Module-Reg...	12:27
3	11-03-Terraform-Cloud-CLI-Driven-Workflow	
4	11-04-Migrate-State-to-Terraform-Cloud	
5		
6		
7		
8		
9		
10		
11		
12		
13		

^ Terraform Cloud & Enterprise Capabilities 13 lectures • 2hr 15min

- ▶ Step-02: Setup Github Repository Local & Remote
- ▶ Step-03: Create Terraform Organization, Workspace and Input Variables

Terraform Cloud Version Control Workflow



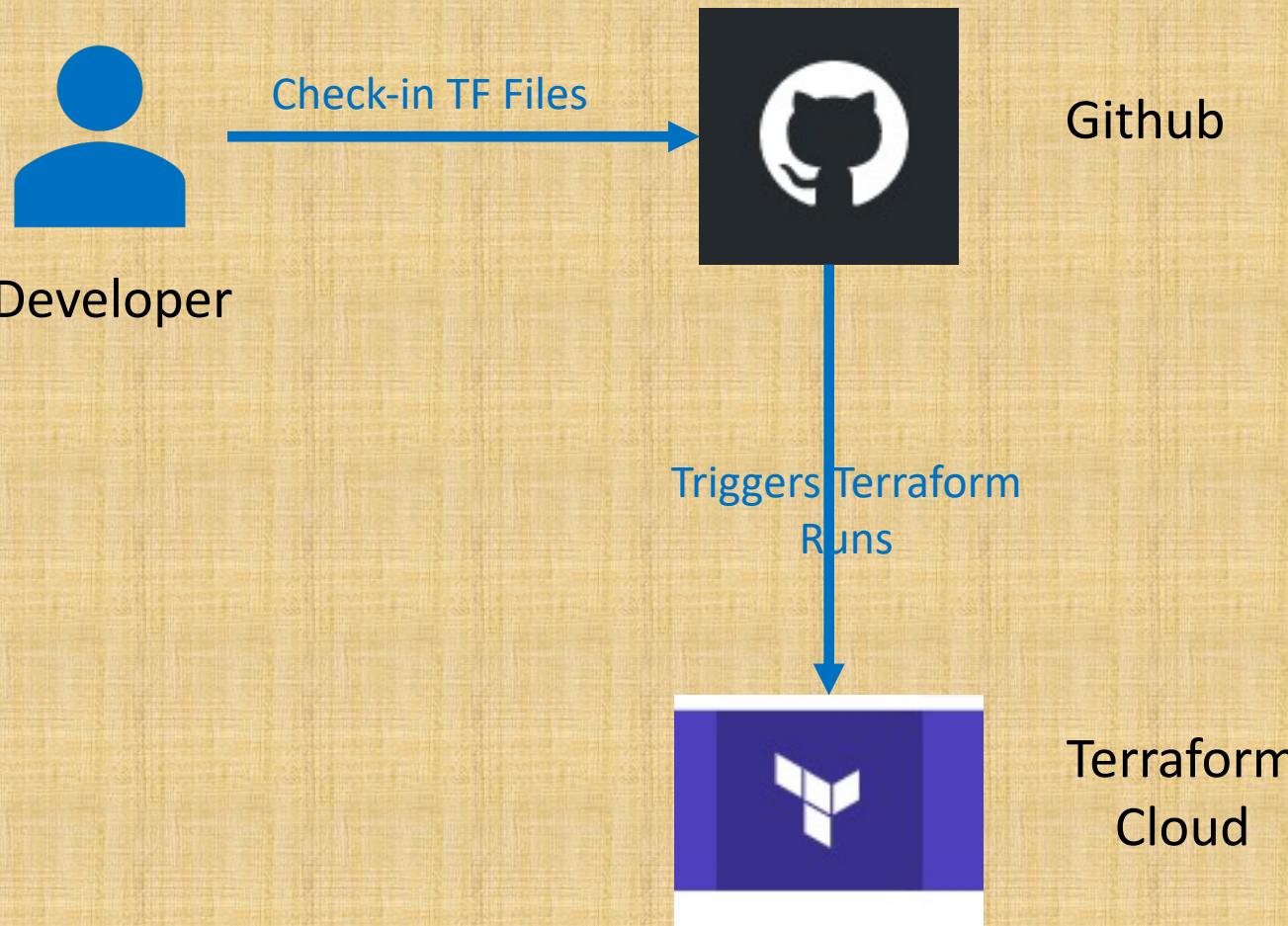
Terraform VCS Integration

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.



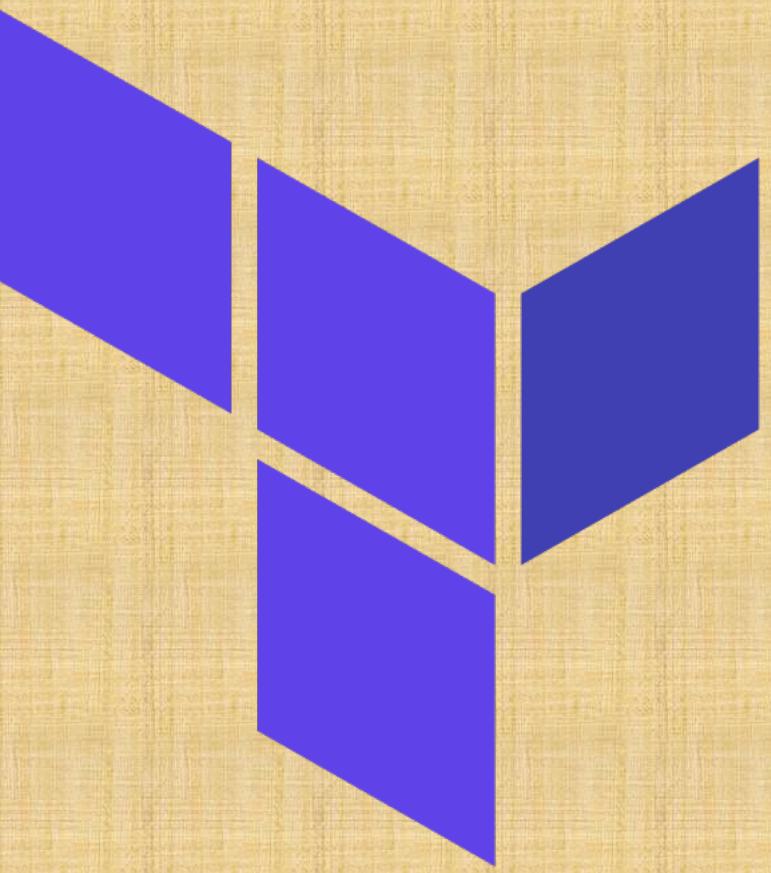
Terraform Cloud - VCS Workflow

The screenshot shows the Terraform Cloud interface. At the top, there's a navigation bar with a back arrow, forward arrow, refresh button, and a URL bar containing `app.terraform.io/app/hcta-demo1/workspaces/terraform-cloud-demo1/runs`. The main menu includes 'Workspaces' (which is highlighted), 'Modules', 'Settings', and 'HashiCorp Cloud Platform'. On the right side of the header are a help icon, a user profile icon, and a dark mode switch.

The main content area shows the 'hcta-demo1 / Workspaces / terraform-cloud-demo1 / Runs' path. Below this, the workspace name 'terraform-cloud-demo1' is displayed with a tooltip. To the right of the workspace name are tabs for 'Runs' (which is selected), 'States', 'Variables', and 'Settings'. Further right is a 'Queue plan' dropdown with a three-line icon.

The 'Current Run' section features a card for 'CIS-Policies-Test-V2'. The card has a small icon, the run name, a 'CURRENT' status indicator, and a 'DISCARDED' button with a cross icon. The background of the workspace page is light gray.

Terraform Cloud Private Module Registry

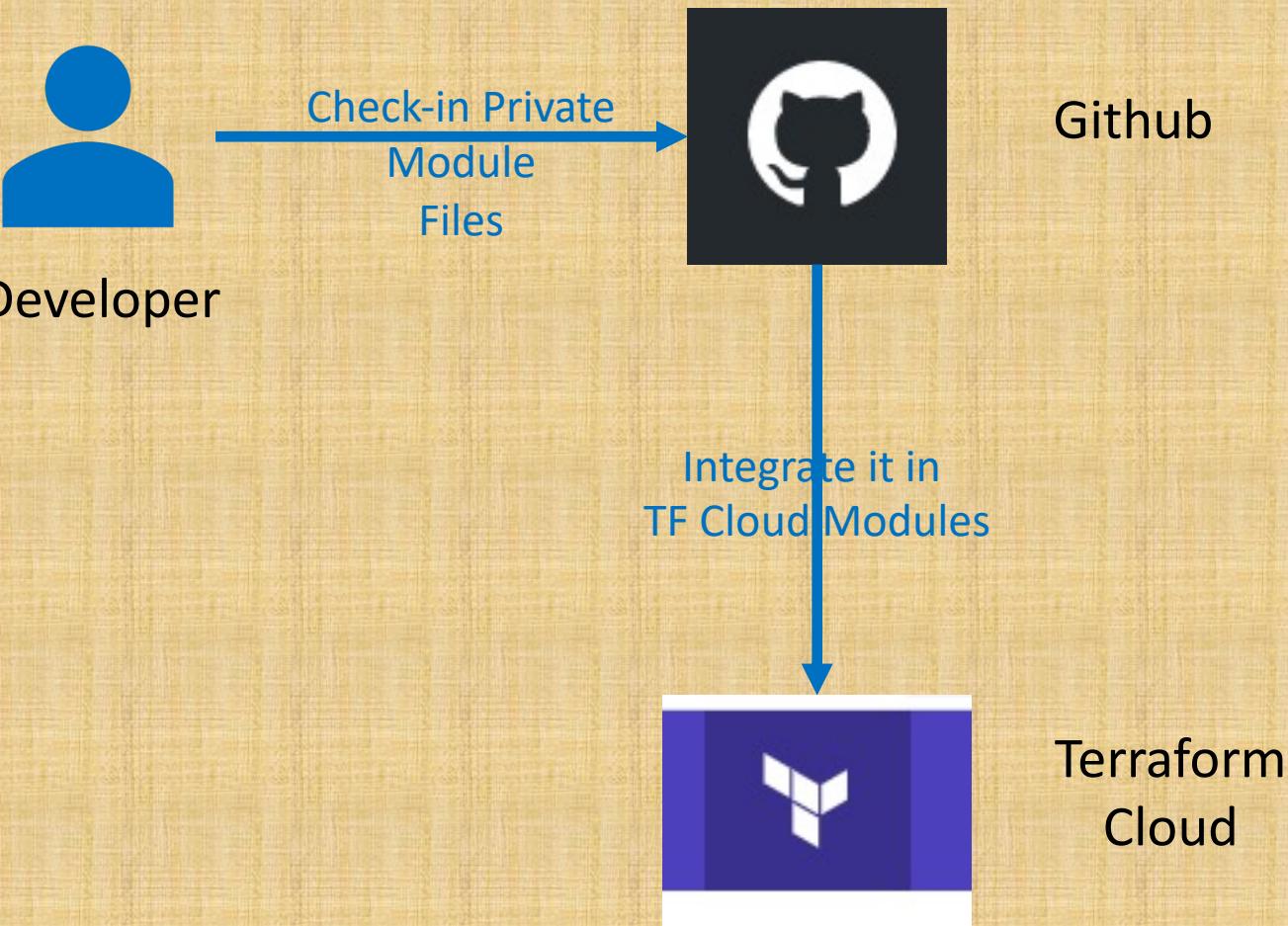


Terraform Private Module Registry

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.

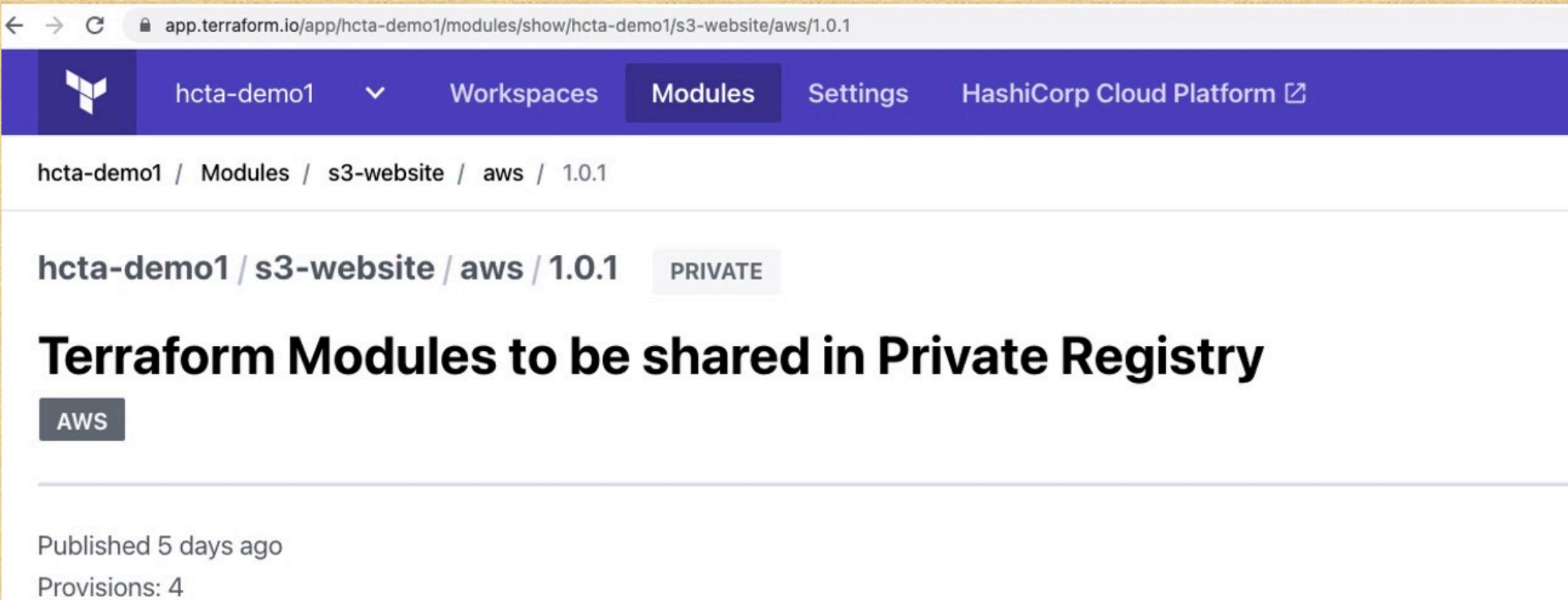
By design, the private module registry works much like the public Terraform Registry. If you're already used the public registry, Terraform Cloud's registry will feel familiar.



Terraform Cloud – Private Module Registry

The screenshot shows the Terraform Cloud interface for managing private modules. The top navigation bar includes links for 'hcta-demo1', 'Workspaces', 'Modules' (which is the active tab), 'Settings', and 'HashiCorp Cloud Platform'. Below the navigation, the path 'hcta-demo1 / Modules' is displayed. On the right side of the header, there are buttons for '+ Design configuration' and '+'. A search bar at the top contains the text 'consul'. To the right of the search bar are a magnifying glass icon and a 'Providers' dropdown menu. The main content area displays a module card for 's3-website'. The card indicates it is a 'PRIVATE' module and describes it as 'Terraform Modules to be shared in Private Registry'. It features an 'AWS' provider badge and 'Version 1.0.1'. A 'Details' button is located in the bottom right corner of the card.

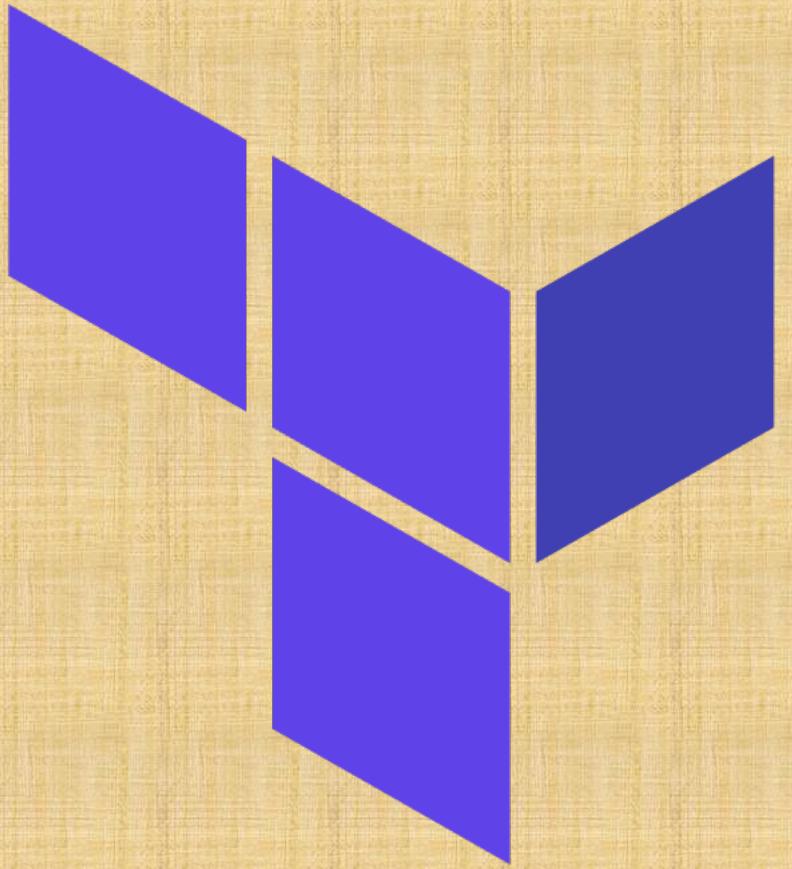
Terraform Cloud – Private Module Registry



The screenshot shows a web browser window for the Terraform Cloud private module registry. The URL in the address bar is `app.terraform.io/app/hcta-demo1/modules/show/hcta-demo1/s3-website/aws/1.0.1`. The page has a purple header with the HashiCorp logo, workspace name "hcta-demo1", navigation links for "Modules", "Workspaces", "Settings", and "HashiCorp Cloud Platform". The main content area displays the module details: "hcta-demo1 / s3-website / aws / 1.0.1" and a "PRIVATE" status badge. The title "Terraform Modules to be shared in Private Registry" is prominently displayed. A "AWS" category badge is present. Below the title, it says "Published 5 days ago" and "Provisions: 4".

Terraform Cloud

CLI Driven Workflow



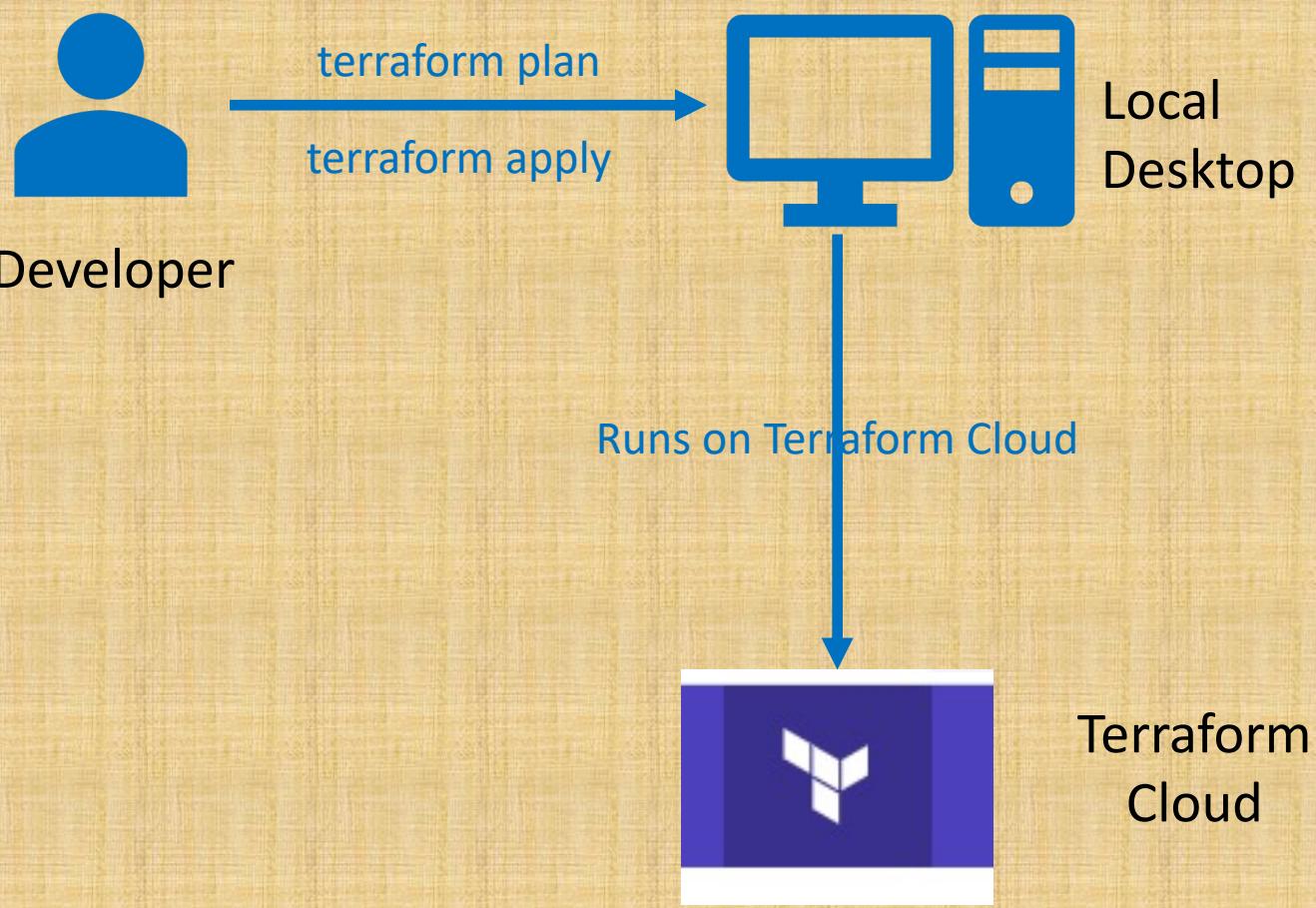
Terraform Cloud - CLI Driven Workflow

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.

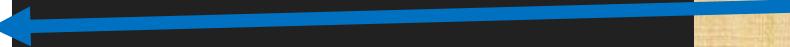


Terraform Cloud - CLI Driven Workflow

```
# Terraform Block
terraform {
  required_version = "~> 0.14" # which means any version equal to or greater than 0.14
  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

Terraform Cloud - CLI Driven Workflow

The screenshot shows the HashiCorp Cloud Platform interface for a workspace named 'cli-driven-demo'. The navigation bar includes 'Workspaces' (selected), 'Modules', 'Settings', and 'HashiCorp Cloud Platform'. The main content area displays a run titled 'cli-driven-demo' with a status of 'Queued manually using Terraform'. The run history shows three steps: 'stacksimplify triggered a run from Terraform 4 days ago' (Run Details), 'Plan finished' (4 days ago, Resources: 0 to add, 0 to change, 1 to destroy), and 'Apply finished' (4 days ago, Resources: 0 added, 0 changed, 1 destroyed). A 'Queue plan' button is visible in the top right.

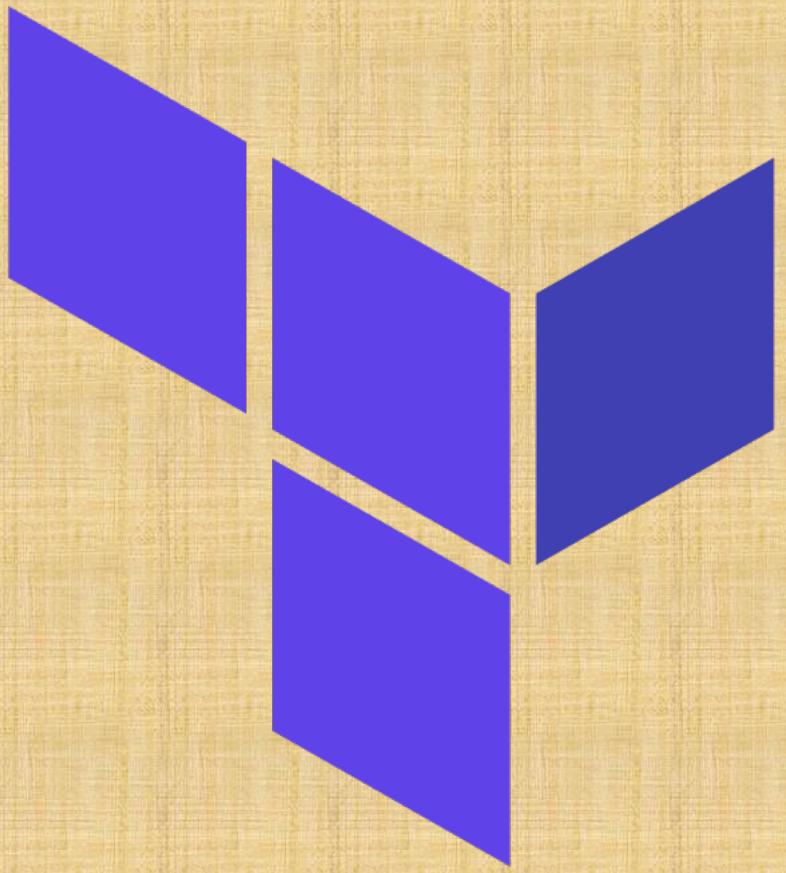
Queued manually using Terraform

stacksimplify triggered a run from Terraform 4 days ago

Plan finished 4 days ago

Apply finished 4 days ago

Terraform Cloud Sentinel



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.

Sentinel – Basic Policies

The screenshot shows a Terraform run history for a workspace named "sentinel-demo1". The status bar at the top indicates "APPLIED" and "Queued manually using Terraform". The history log shows three successful steps: "Plan finished" (3 days ago), "Cost estimation finished" (3 days ago), and "Policy check passed" (3 days ago). Below the log, two policy results are listed: "passed terraform-sentinel-policies/check-terraform-version" and "passed terraform-sentinel-policies/restrict-s3-buckets". A blue arrow points from the "Policy check passed" step in the log to a large blue circle containing the text "Sentinel Policies".

sentinel-demo1 ⓘ

✓ APPLIED Queued manually using Terraform

✓ Plan finished 3 days ago

✓ Cost estimation finished 3 days ago

✓ Policy check passed 3 days ago

Passed 3 days ago Passed 3 days ago

✓ passed terraform-sentinel-policies/check-terraform-version

✓ passed terraform-sentinel-policies/restrict-s3-buckets

Sentinel
Policies

Sentinel – Cost Control Policies

The screenshot shows a list of events from a Terraform Cloud run. The first three events are green boxes indicating success: 'Plan finished' (3 days ago), 'Cost estimation finished' (3 days ago), and 'Policy check passed' (3 days ago). Below these, the status 'Queued' is shown followed by an arrow pointing to 'Passed' (3 days ago). At the bottom, a green checkmark indicates a policy named 'terraform-sentinel-cost-control-policies/less-than-100-month' has passed.

→ C app.terraform.io/app/hcta-demo1/workspaces/terraform-cloud-demo1/runs/run-CLdshzPSQ51QY1UT

Plan finished 3 days ago

Cost estimation finished 3 days ago

Policy check passed 3 days ago

Queued 3 days ago > Passed 3 days ago

passed **terraform-sentinel-cost-control-policies/less-than-100-month**

Sentinel – CIS Policies

The screenshot shows a Terraform Cloud run history with the following steps:

- Plan finished** 3 days ago (green checkmark)
- Cost estimation finished** 3 days ago (green checkmark)
- Policy check passed** 3 days ago (green checkmark)

Below the policy check section, the status is listed as **Queued 3 days ago > Passed 3 days ago**.

Details of the passed policies:

- advisory failed** terraform-sentinel-cis-policies/aws-cis-4.1-networking-deny-public-ssh-acl-rules (orange exclamation mark)
- passed** terraform-sentinel-cis-policies/aws-cis-4.2-networking-deny-public-rdp-acl-rules (green checkmark)
- passed** terraform-sentinel-cis-policies/aws-cis-4.3-networking-restrict-all-vpc-traffic-acl-rules (green checkmark)
- passed** terraform-sentinel-cost-control-policies/less-than-100-month (green checkmark)

Practical Examples with Step-by-Step Github Documentation

The screenshot shows a GitHub repository structure. At the top level, there is a folder named "12-Terraform-Cloud-and-Sentinel". Inside this folder, there are three sub-folders: "12-01-Terraform-Cloud-and-Sentinel-Policies", "12-02-Control-Costs-with-Sentinel-Policies", and "12-03-Terraform-Foundational-Policies-using-Sentinel". Each of these sub-folders contains several files: "Sentinel-Mocks", "terraform-manifests", "terraform-sentinel-policies", and a README.md file. The "terraform-sentinel-policies" file is expanded to show its contents, which include "check-terraform-version.sentinel", "restrict-s3-buckets.sentinel", and "sentinel.hcl". The "12-03-Terraform-Foundational-Policies-using-Sentinel" folder is also expanded to show its "terraform-sentinel-cis-policies" file, which contains "sentinel.hcl" and a README.md file.

- ✓ 12-Terraform-Cloud-and-Sentinel
 - ✓ 12-01-Terraform-Cloud-and-Sentinel-Policies
 - > Sentinel-Mocks
 - > terraform-manifests
 - ✓ terraform-sentinel-policies
 - ≡ check-terraform-version.sentinel
 - ≡ restrict-s3-buckets.sentinel
 - ≡ sentinel.hcl
 - ⓘ README.md
 - ✓ 12-02-Control-Costs-with-Sentinel-Policies
 - > Sentinel-Mocks
 - > terraform-manifests
 - ✓ terraform-sentinel-cost-control-policies
 - ≡ less-than-100-month.sentinel
 - ≡ sentinel.hcl
 - ⓘ README.md
 - ✓ 12-03-Terraform-Foundational-Policies-using-Sentinel
 - ✓ terraform-sentinel-cis-policies
 - ≡ sentinel.hcl
 - ⓘ README.md

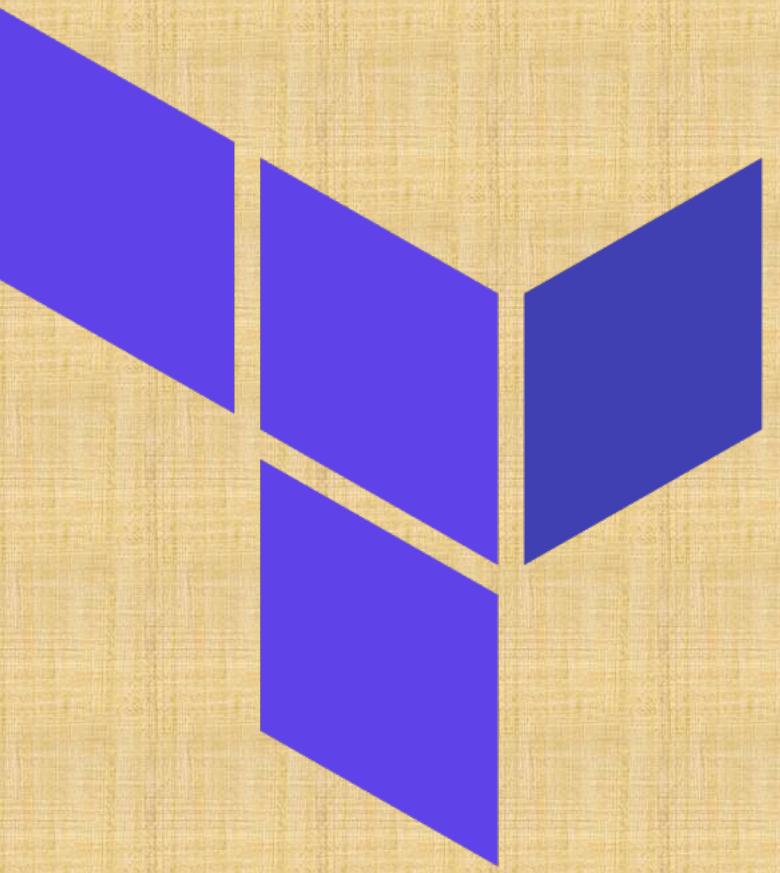
The screenshot shows a video player interface. At the top, there is a navigation bar with "Welcome to" followed by three course sections: "12-01-Terraform-Cloud-and-Sentinel-Policies", "12-02-Control-Costs-with-Sentinel-Policies", and "12-03-Terraform-Foundational-Policies-using-Sentinel". Below this, there is a section titled "Terraform Cloud & Sentinel" with a duration of "13 lectures • 1hr 48min". The video player shows two steps: "Step-02: Review Terraform Manifests used as part of this Demo" (duration 05:05) and "Step-03: Understand Terraform Free and Paid Plan Features and Enable Trial for T" (duration 07:04). The first step is highlighted with a green border.

- ..
- 12-01-Terraform-Cloud-and-Sentinel-Policies Welcome to
- 12-02-Control-Costs-with-Sentinel-Policies Welcome to
- 12-03-Terraform-Foundational-Policies-using-Sentinel Welcome to

^ Terraform Cloud & Sentinel 13 lectures • 1hr 48min

- ▶ Step-02: Review Terraform Manifests used as part of this Demo 05:05
- ▶ Step-03: Understand Terraform Free and Paid Plan Features and Enable Trial for T 07:04

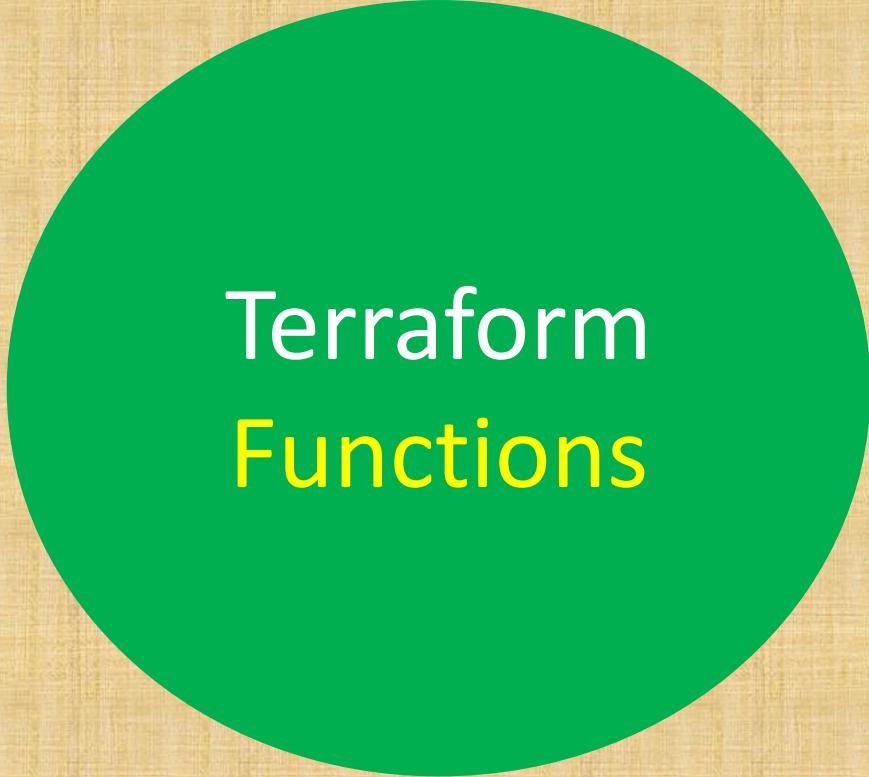
Terraform Expressions



Terraform Expressions

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

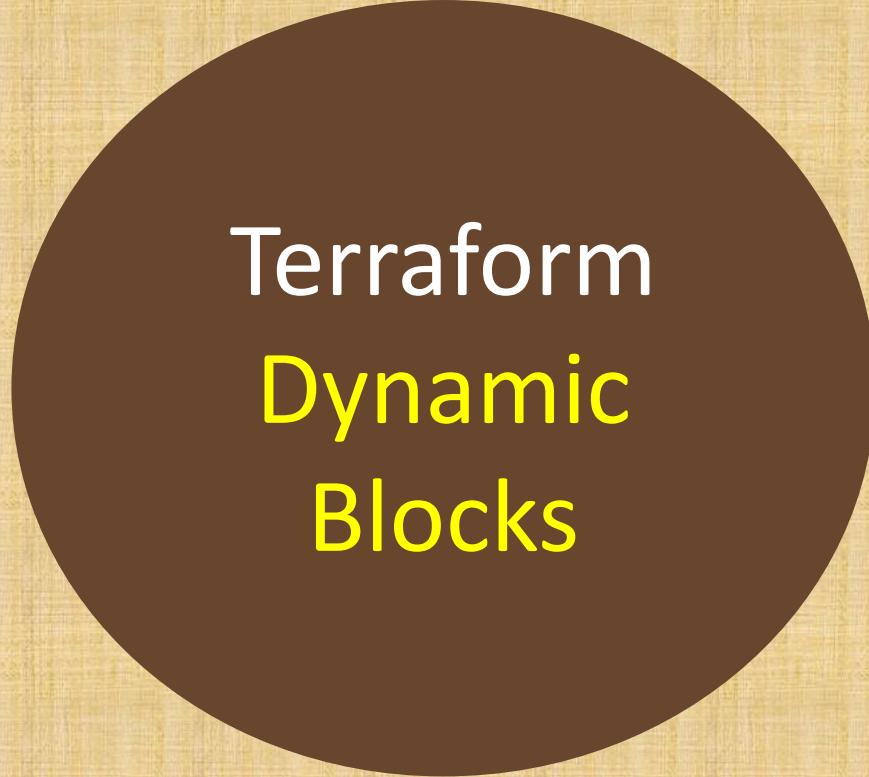
You can experiment with the behavior of Terraform's expressions from the Terraform expression console, by running the terraform console command.



Terraform
Functions

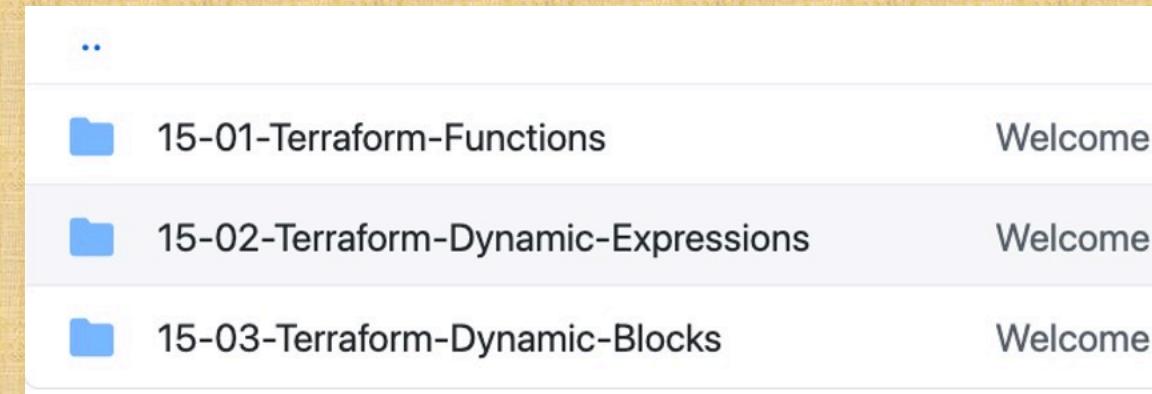
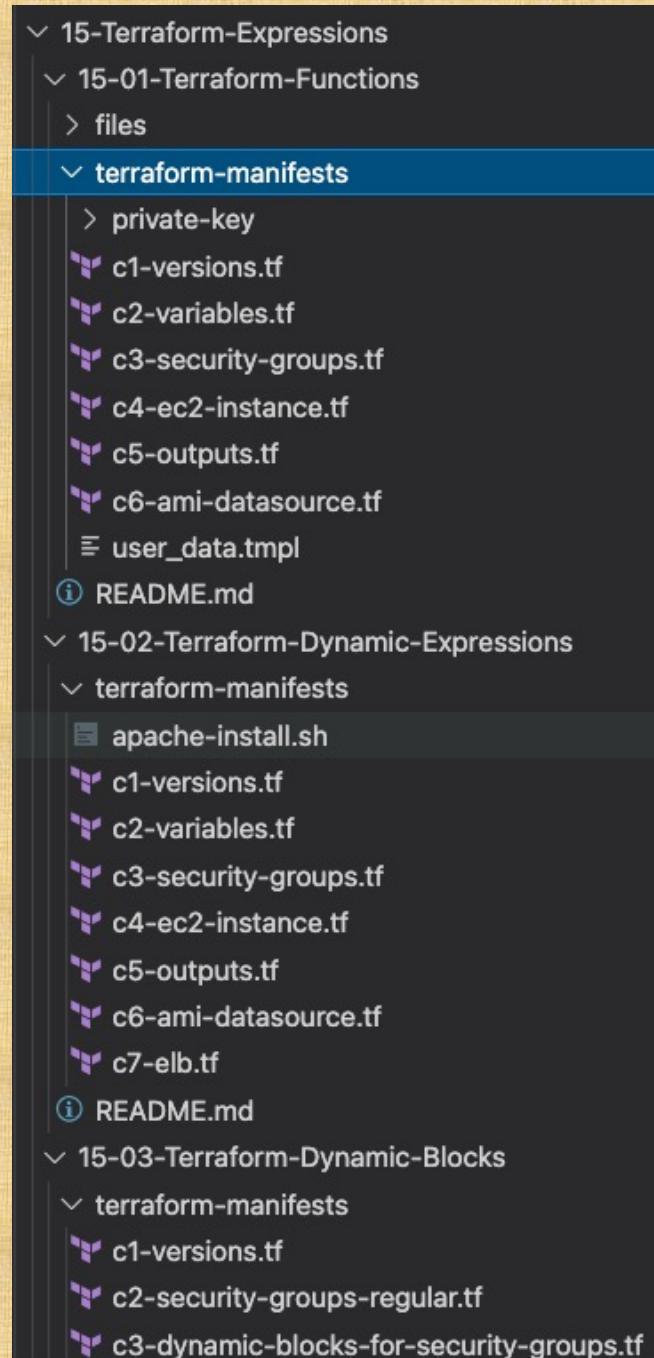


Terraform
Dynamic
Expressions



Terraform
Dynamic
Blocks

Practical Examples with Step-by-Step Github Documentation



A screenshot of a learning platform interface showing a course titled 'Terraform Expressions'.

Terraform Expressions (9 lectures • 1hr 20min)

12:35

05:30

Step-01: Part-1: Learn various Terraform Functions using Terraform Console

Step-02: Part-2: Learn various Terraform Functions using Terraform Console

ALL LIVE SLIDES ARE BEFORE THIS SLIDE



Thank You