HashiCorp

Terraform

Greens Technologys

Bharath Waj K S

Day-1

Agenda

- What is IAC ?
- What is Terraform ?
- Why Terraform ?
- Terraform Architecture
- Terraform Advantage
- Terraform Installation
- Terraform Demo

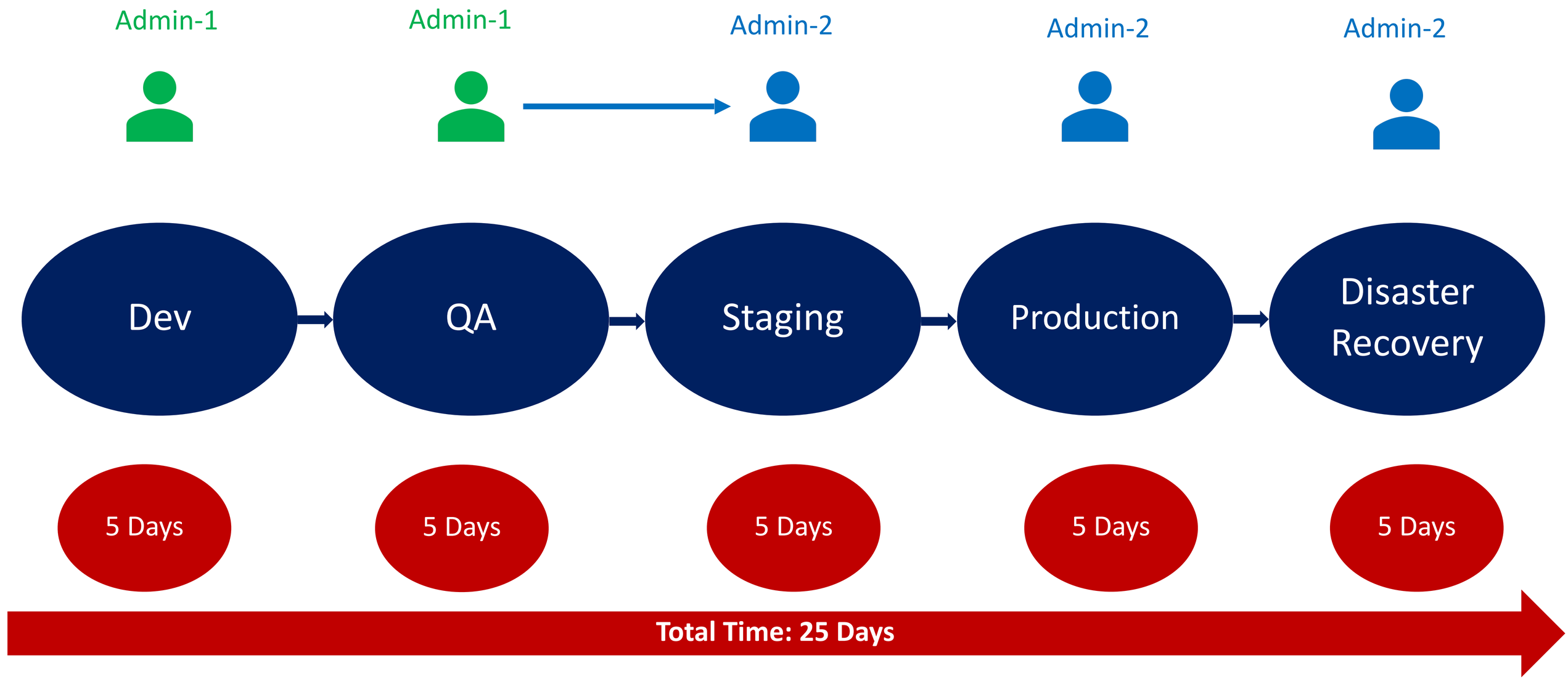
Greens Technologys

What we'll talk about today

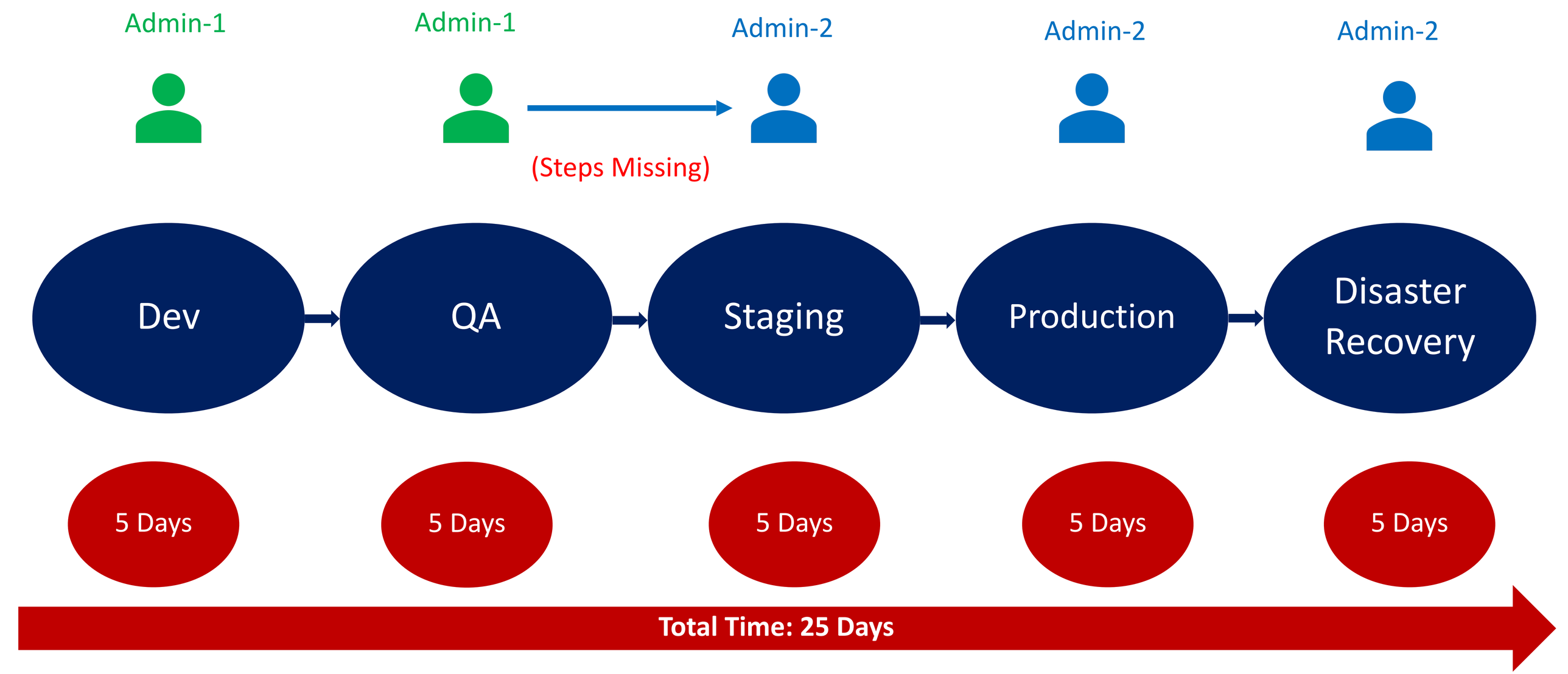
- What's Infrastructure As Code (IaC)
- What is Terraform and how does it work
- Why Terraform
- Terraform Architecture
- Terraform Basic Commands
- Demo

Let’s talk about : Infrastructure as Code

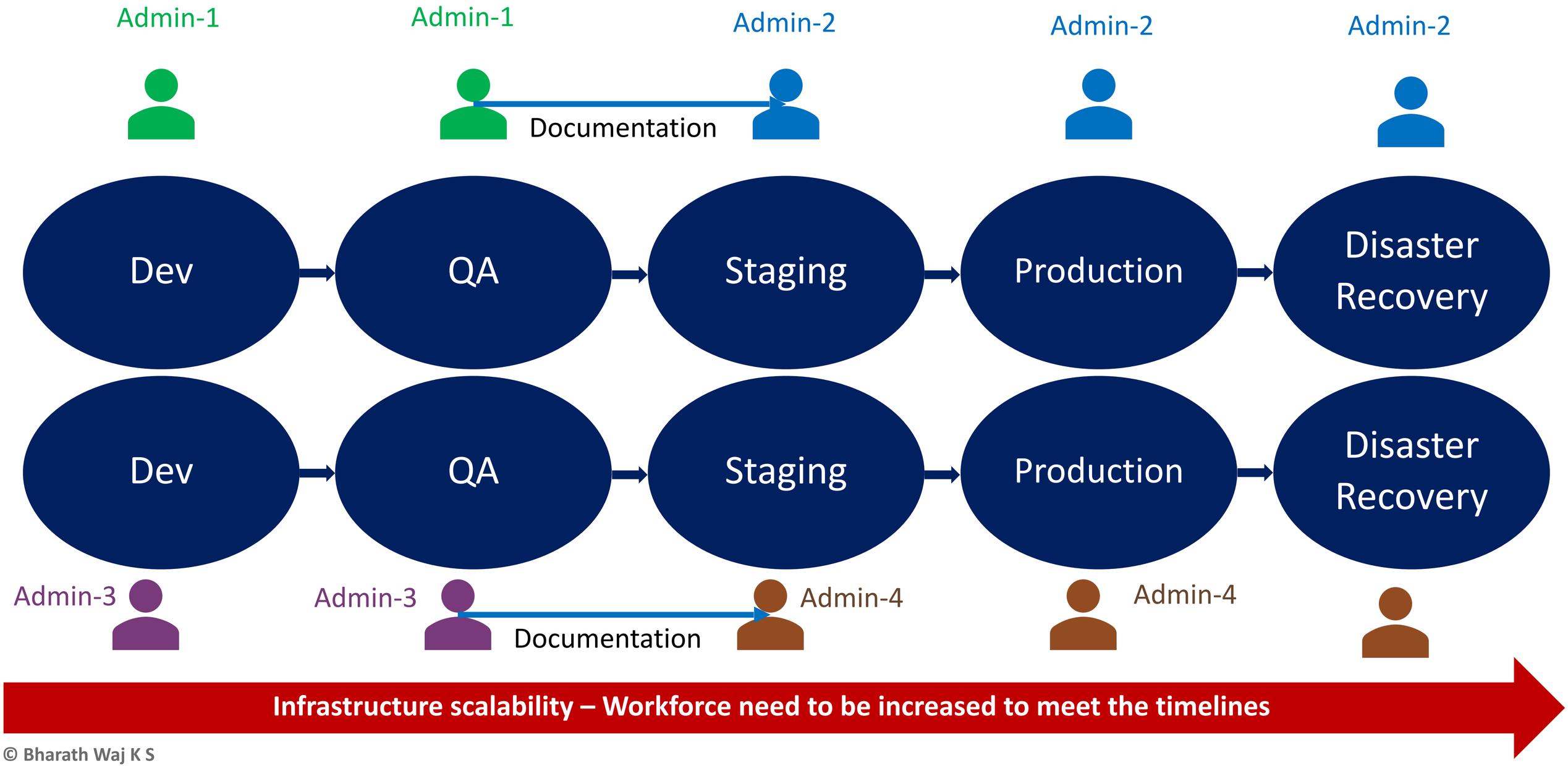
Traditional Way of Managing Infrastructure



Traditional Way of Managing Infrastructure



Traditional Way of Managing Infrastructure



Manage using IaC with Terraform

5 Days

Admin-1

Check-In TF Code

Github

Triggers
TF Runs

Terraform
Cloud

DevOps / CI CD for IaC

Scale-Up and Scale-Down On-Demand

Creates Infra

Dev → QA → Staging → Production → Disaster Recovery

One-Time Work

Re-Use Templates

Quick & Fast

Reliable

Tracked for Audit

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

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.

Advantages of IaC

It’s code, so you can work on it just like your application

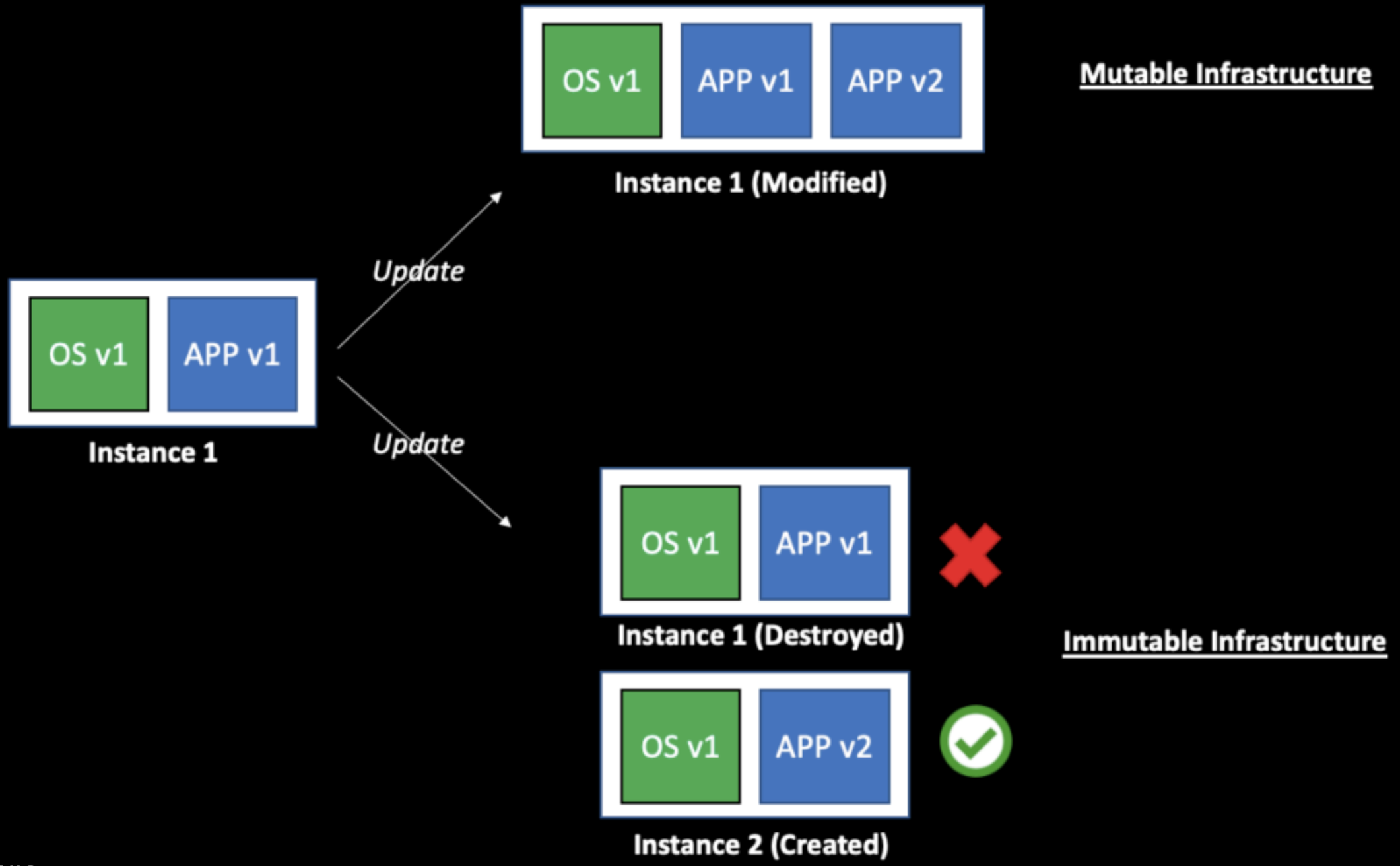
- Versioning
- Easy to collaborate on
- Easy to integrate in a CI setup (it’s code after all)
- Automate your deployment and recovery processes
- No need to repair, just redeploy
- Etc.

The Benefits

Auditable	Repeatable	Dependable
It's easier to audit code than physical infrastructure	Because going through hundreds of different buttons and screens on your cloud provider's web UI isn't the kind of thing that's easy to replicate 100% accurately	If you can’t audit and repeat the actions that built your infrastructure, you can’t depend on your infra

Why Terraform

		
AWS	Mainly Config Management	Mainly IAAC code with 1000 + Providers
JSON	Yaml	HCL
	Procedural (Order / no state / not reusable)	Declarative (No order / aware-state / reusable)
	Mutable (Fast)	Immutable(Slow)

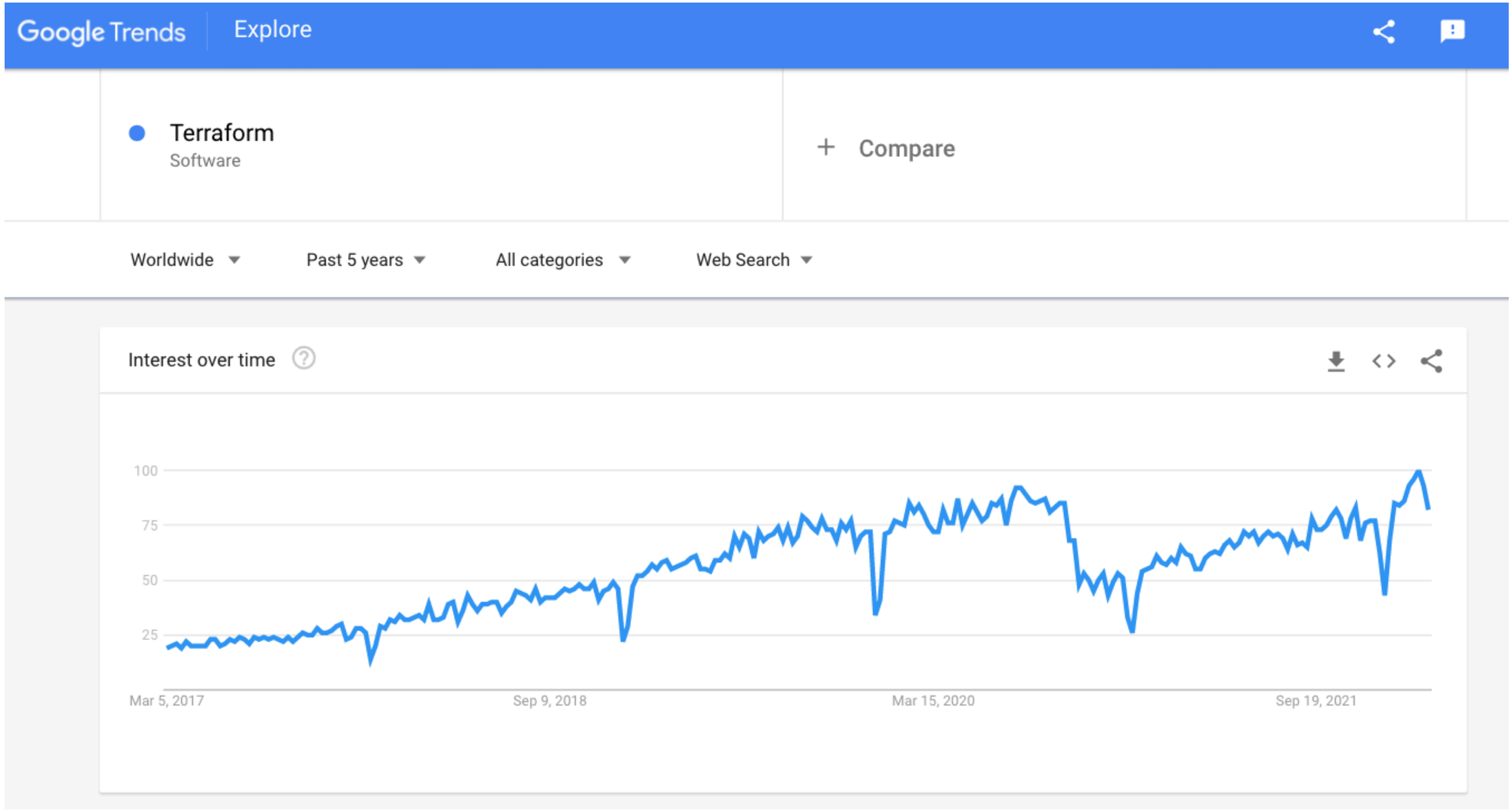


What’s Terraform ?

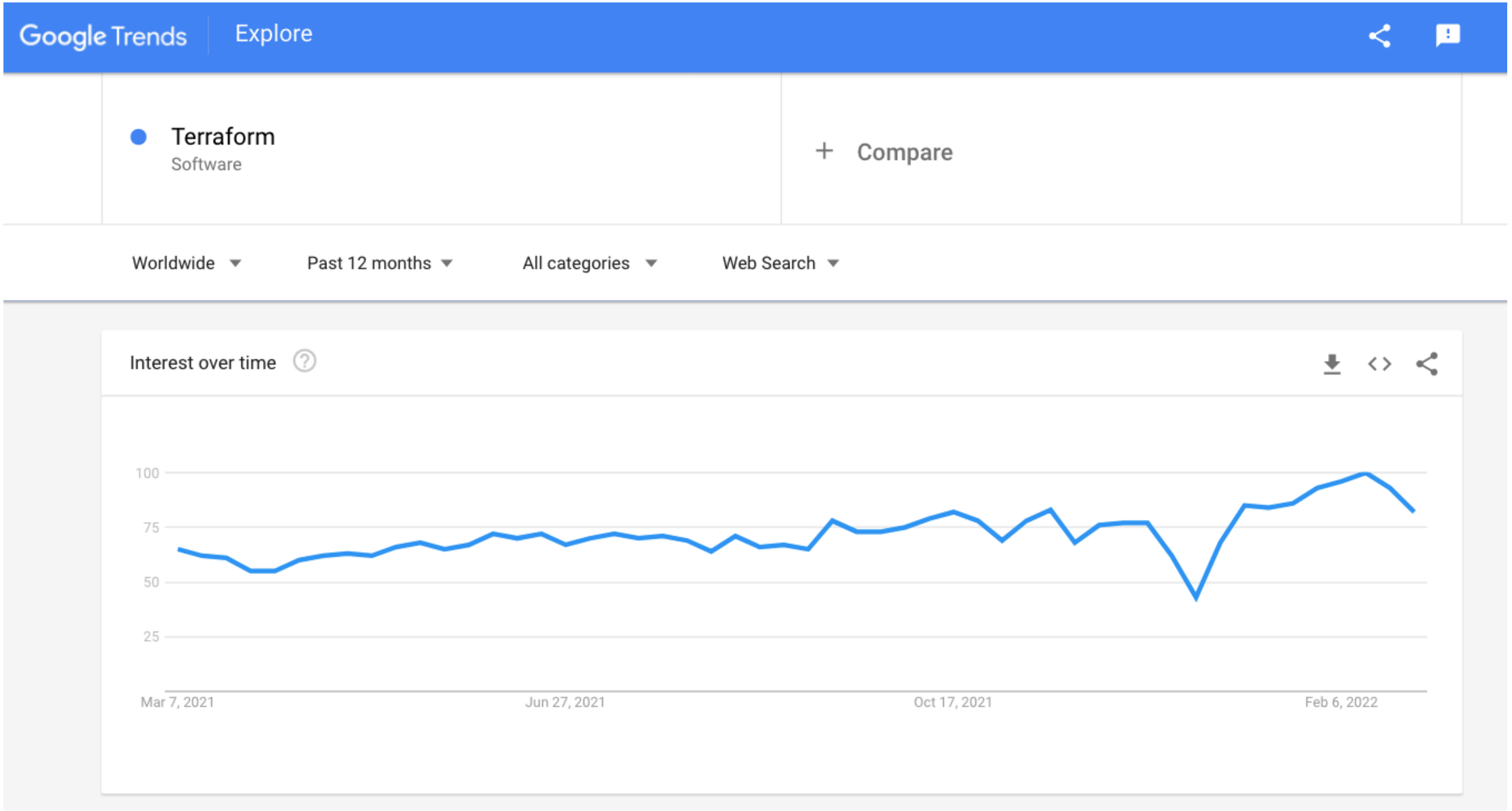
Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.

HashiCorp
Terraform

Google Trends – Past 5 Years

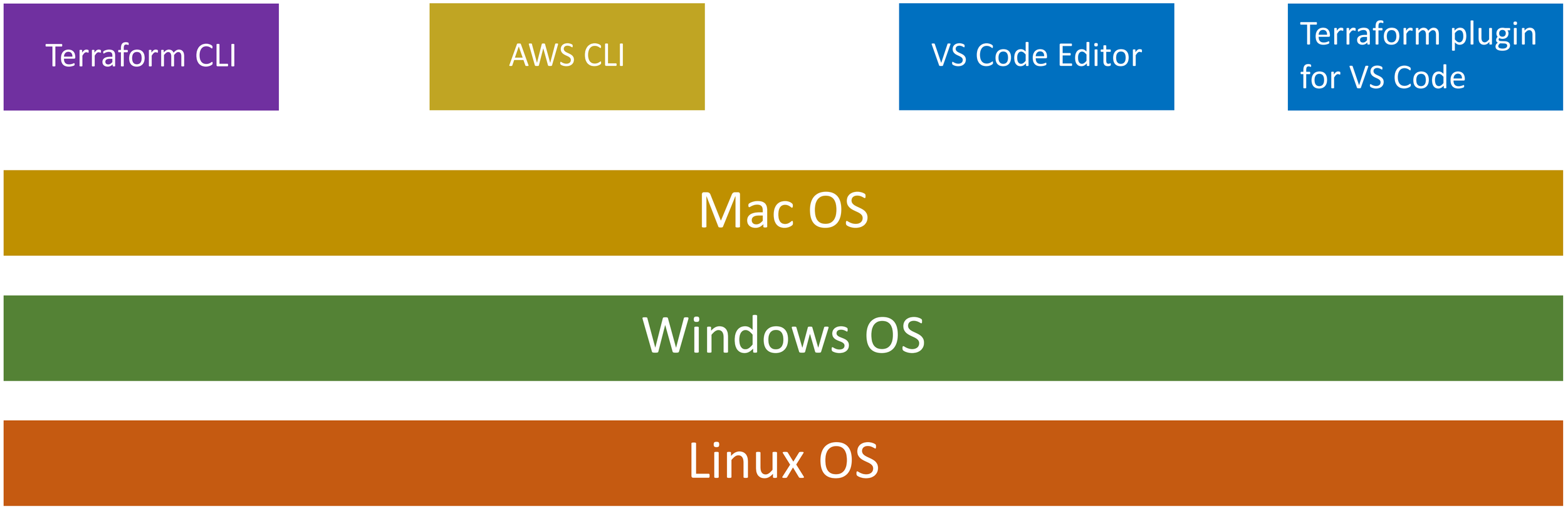


Google Trends – Past 1 Year



Terraform
Installation

Terraform Installation



“DEMO”

–Bharath Waj KS

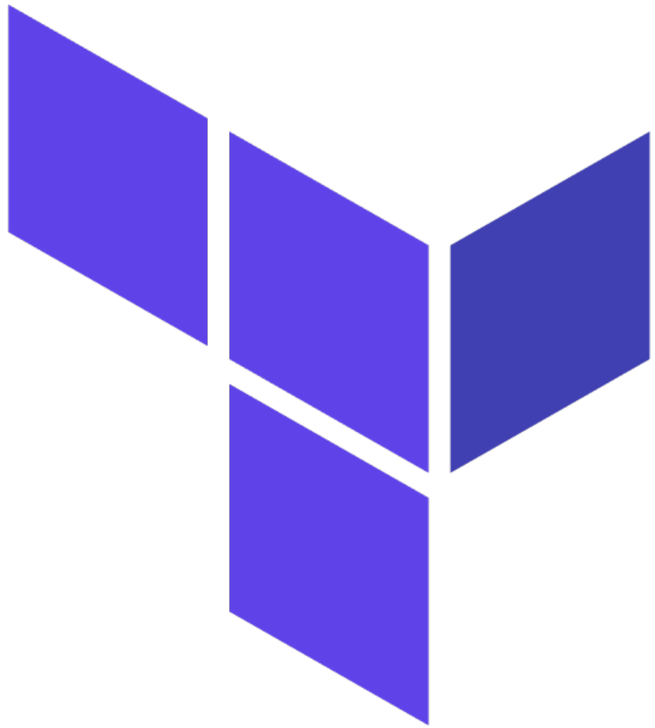
Day-2

Agenda

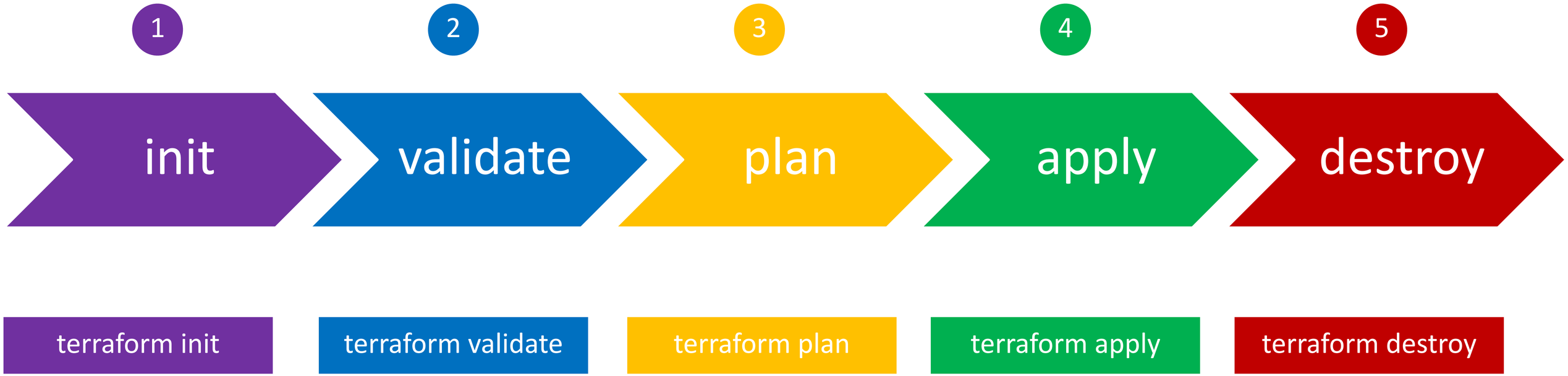
- Terraform Basic Command
- Terraform Block
- Terraform Language Basic
- Terraform Provider
- Terraform Registry
- Terraform Resources
- Terraform Demo

Greens Technologys

Terraform
Command Basics



Terraform Workflow



Terraform Workflow

1 2 3 4 5

init

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

validate

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

plan

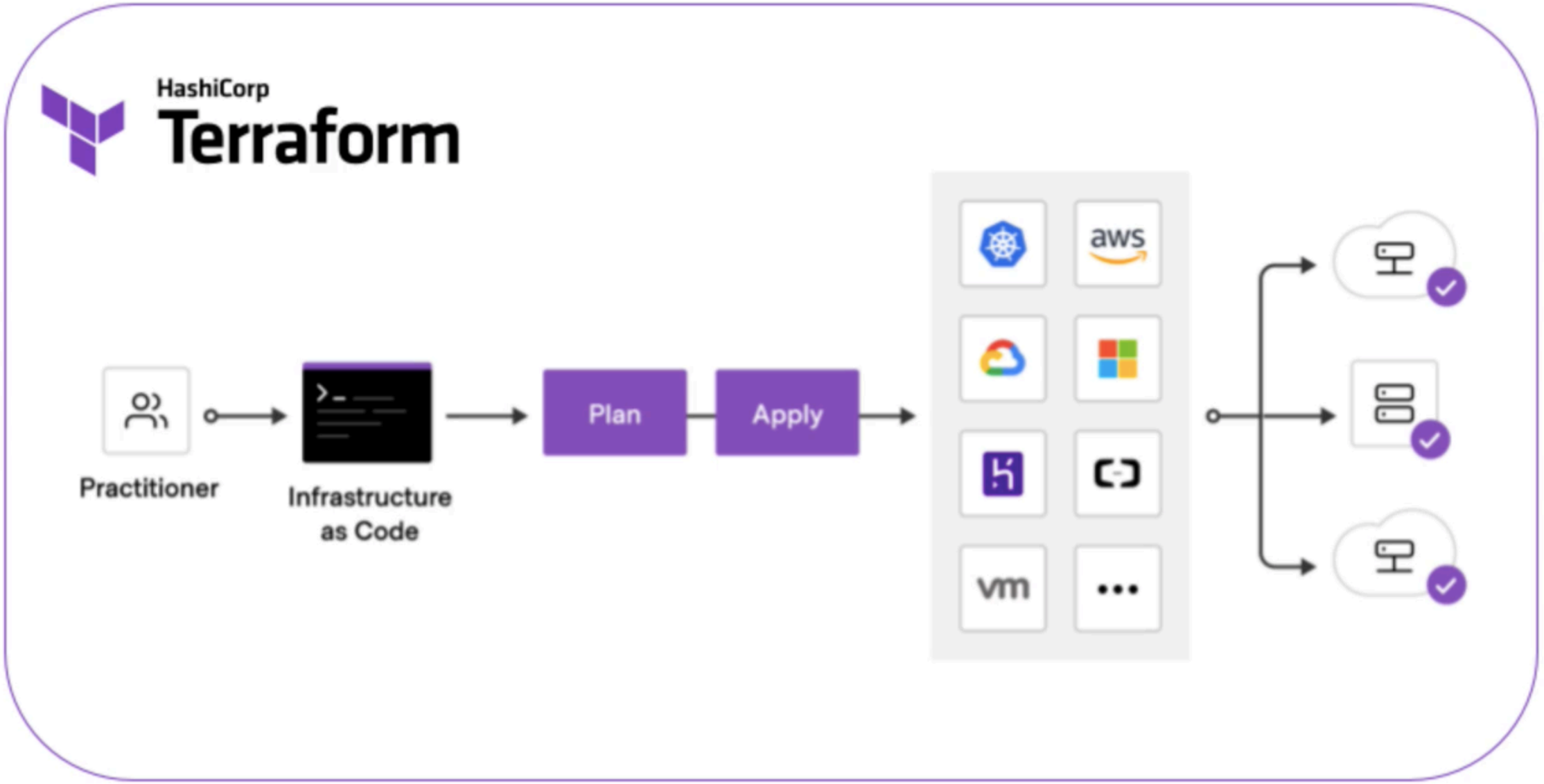
- Creates an execution plan
- Terraform performs a refresh and determines what actions are necessary to achieve the desired state specified in configuration files

apply

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

destroy

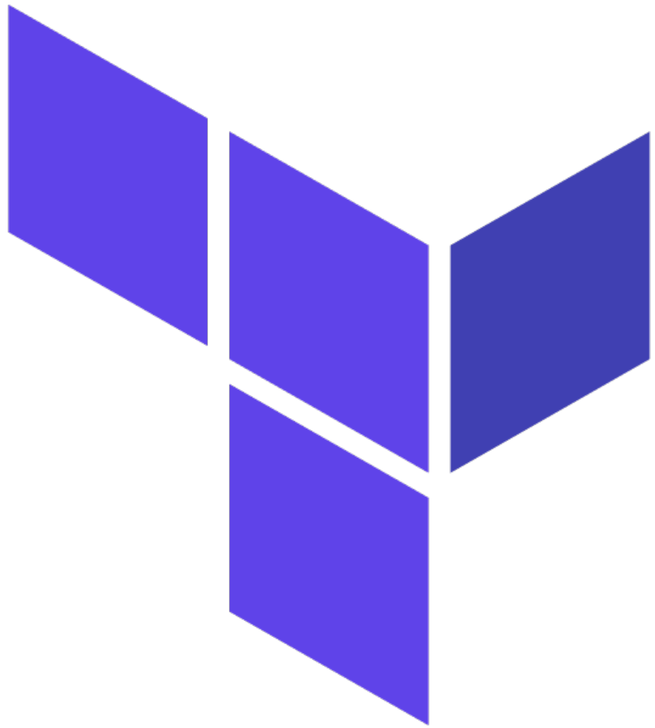
- Used to destroy the Terraform-managed infrastructure
- This will ask for confirmation before destroying.



“DEMO”

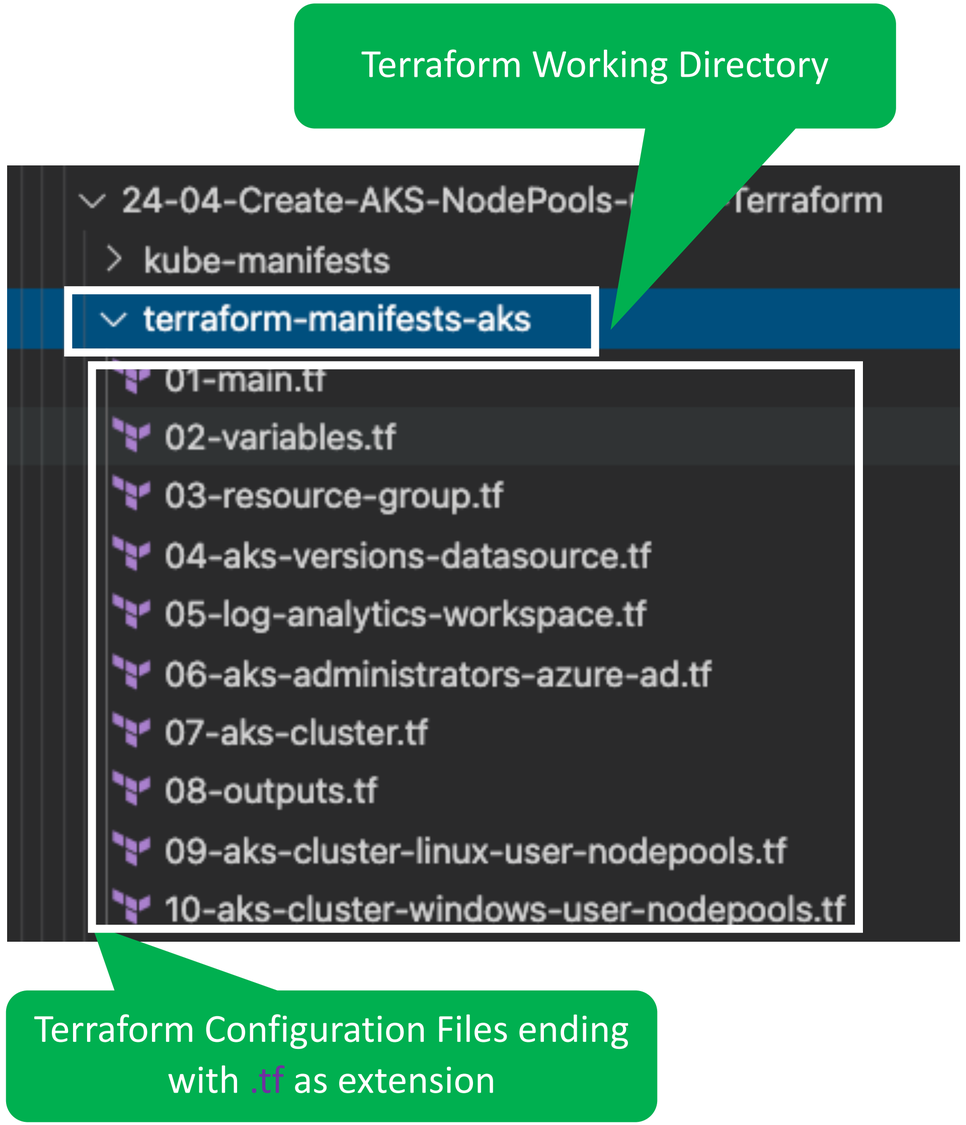
–Bharath Waj KS

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 Language Basics – Configuration Syntax

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

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

Block Labels

Block Type

Top Level & Block inside Blocks

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

Top Level Blocks: resource, provider

Block Inside Block: provisioners, resource specific blocks like tags

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"
}
```

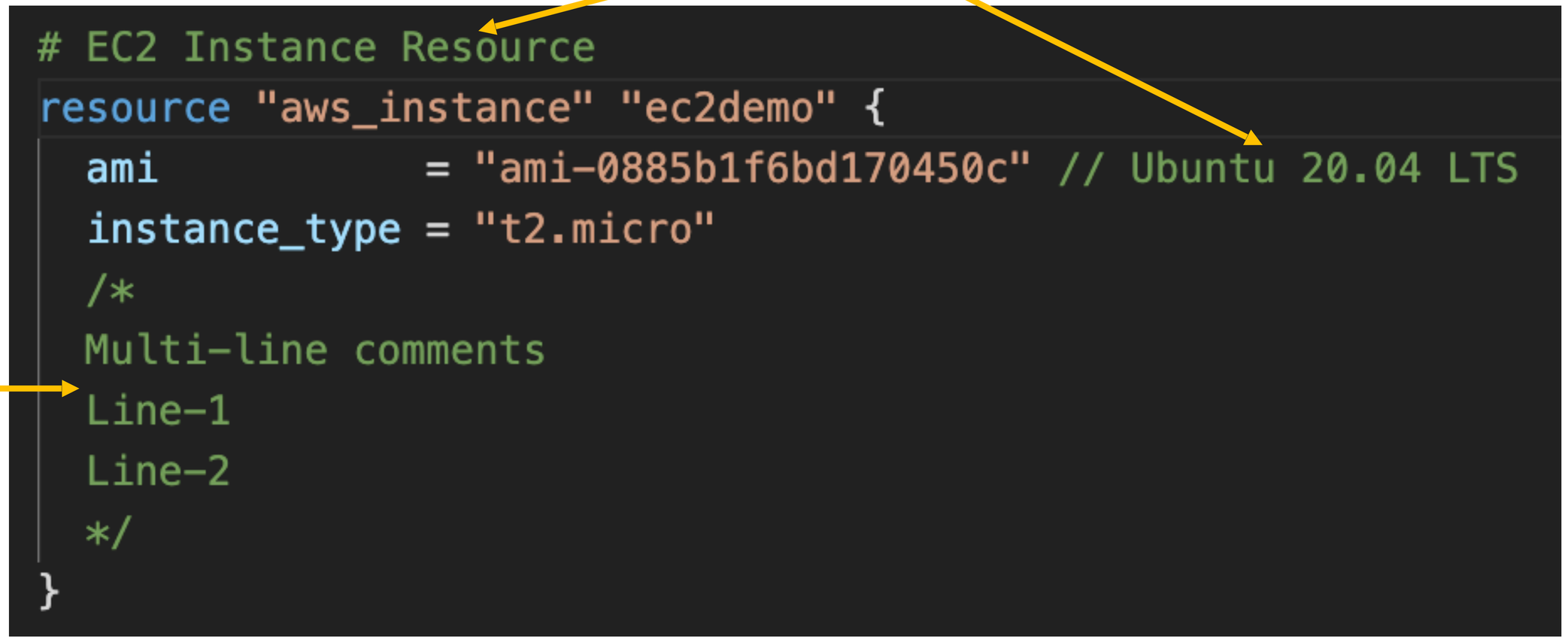
Argument Name
[or]
Identifier

Argument Value
[or]
Expression

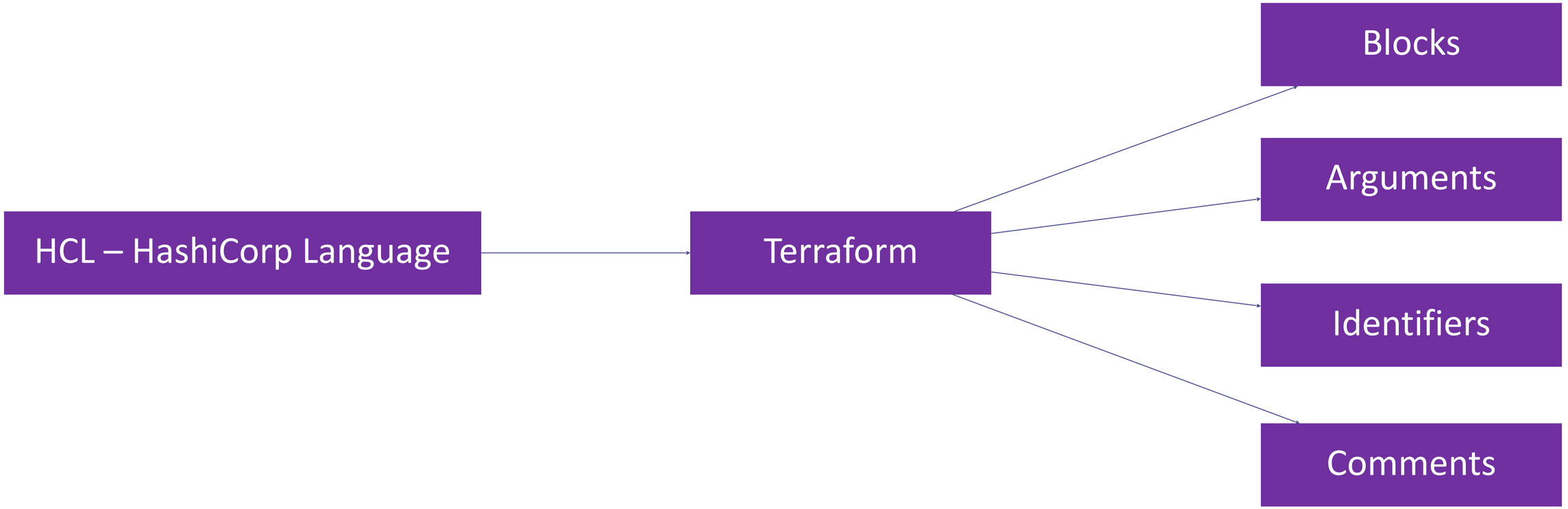
Terraform Language Basics – Configuration Syntax

Single Line Comments with # or //

Multi-line comment



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

Fundamental Blocks

Terraform, Provider, Resources

Terraform Basic Blocks

Terraform Block

- Special block used to configure some behaviors
- Specifying a required Terraform CLI Version
- Specifying Provider Requirements & Versions
- 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.
- **VERY VERY IMPORTANT TO MEMORIZE**
  - 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

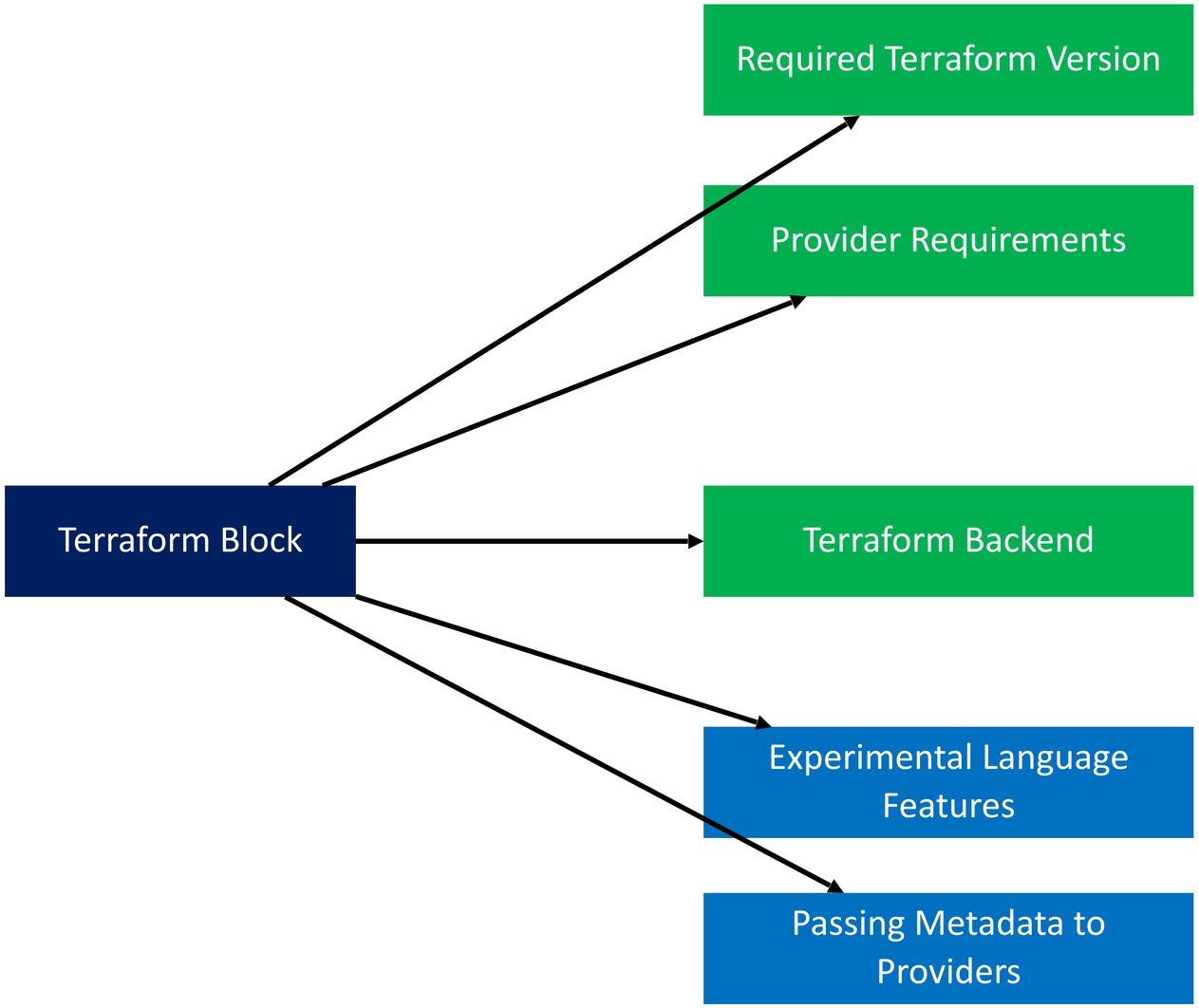
Terraform 0.12 and earlier:

```
# 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:

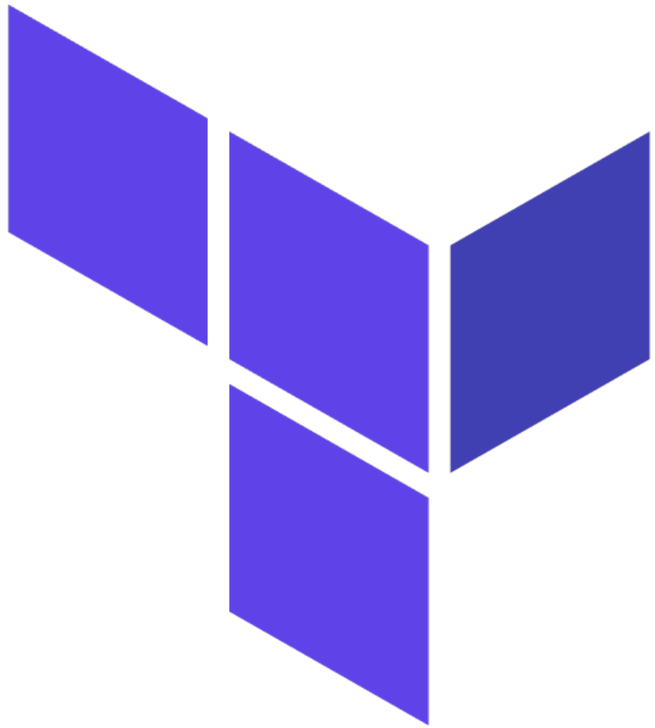
```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# 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




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

Terraform
Providers



Terraform Providers

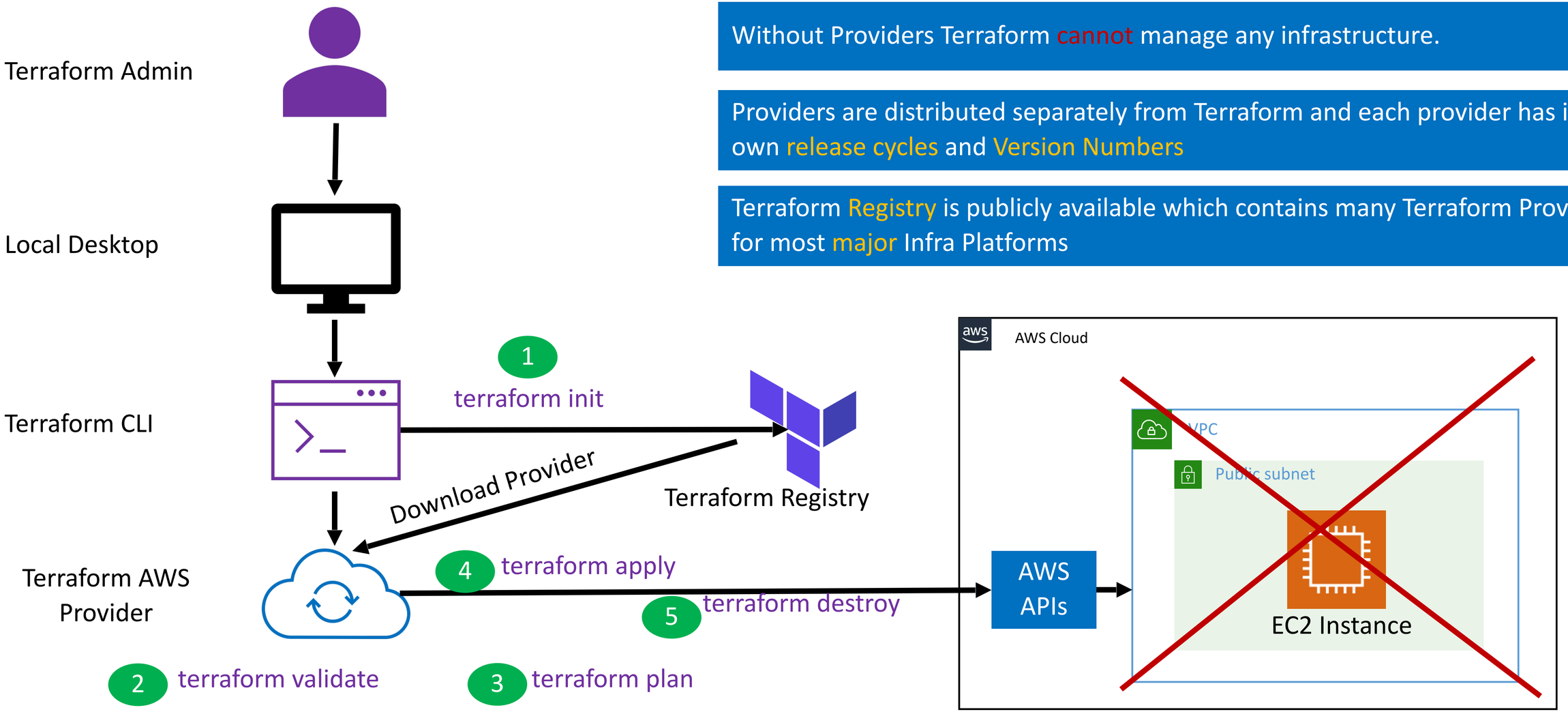
Providers are HEART of Terraform

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

Without Providers Terraform cannot manage any infrastructure.

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

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



Terraform Providers

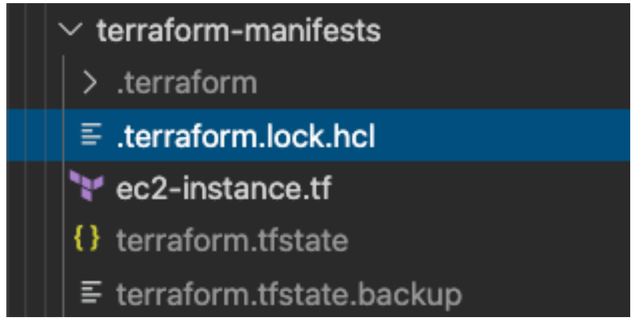
Provider Requirements

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

Provider Configuration

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

Dependency Lock File




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

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

Dependency Lock File

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

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

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

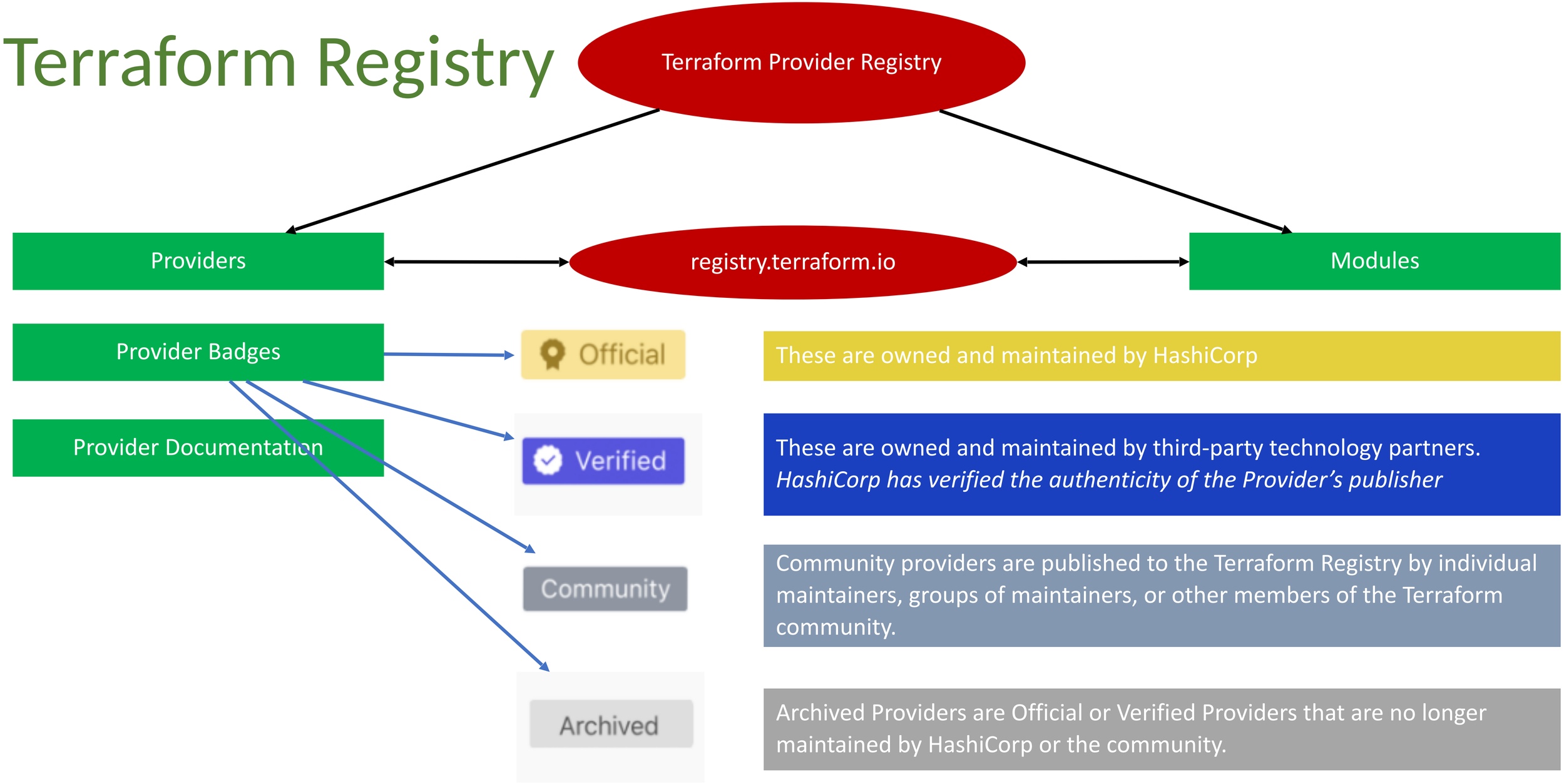
Source addresses consist of three parts delimited by slashes (/)

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

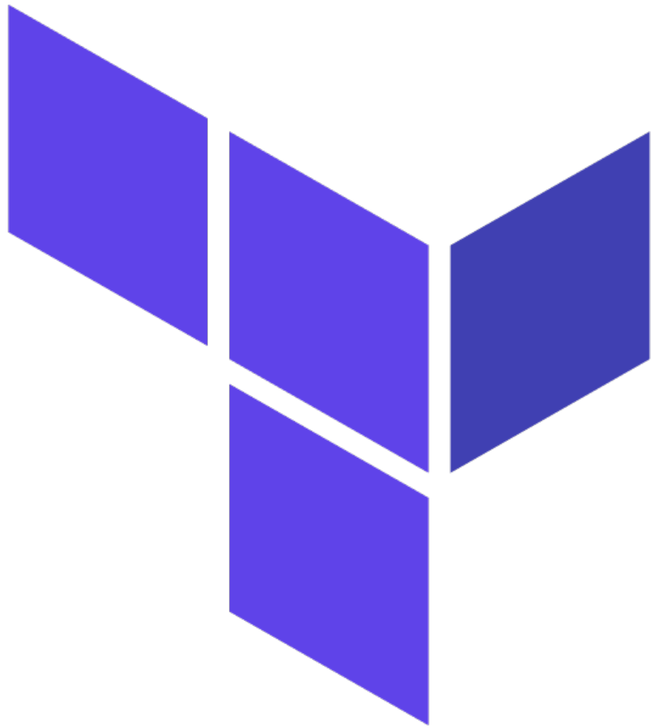
registry.terraform.io/hashicorp/aws

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

Terraform Registry



Terraform Resources Introduction



Terraform Language Basics – Configuration Syntax

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

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

Block Labels

Block Type

Top Level & Block inside Blocks

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

Top Level Blocks: resource, provider

Block Inside Block: provisioners, resource specific blocks like tags

Arguments

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

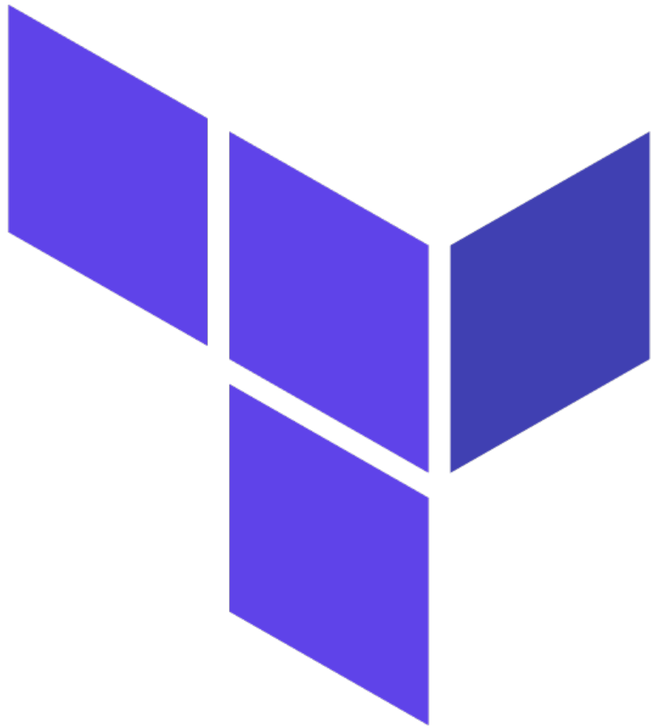
Meta-Arguments: Can be used with any resource to change the behavior of resources

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

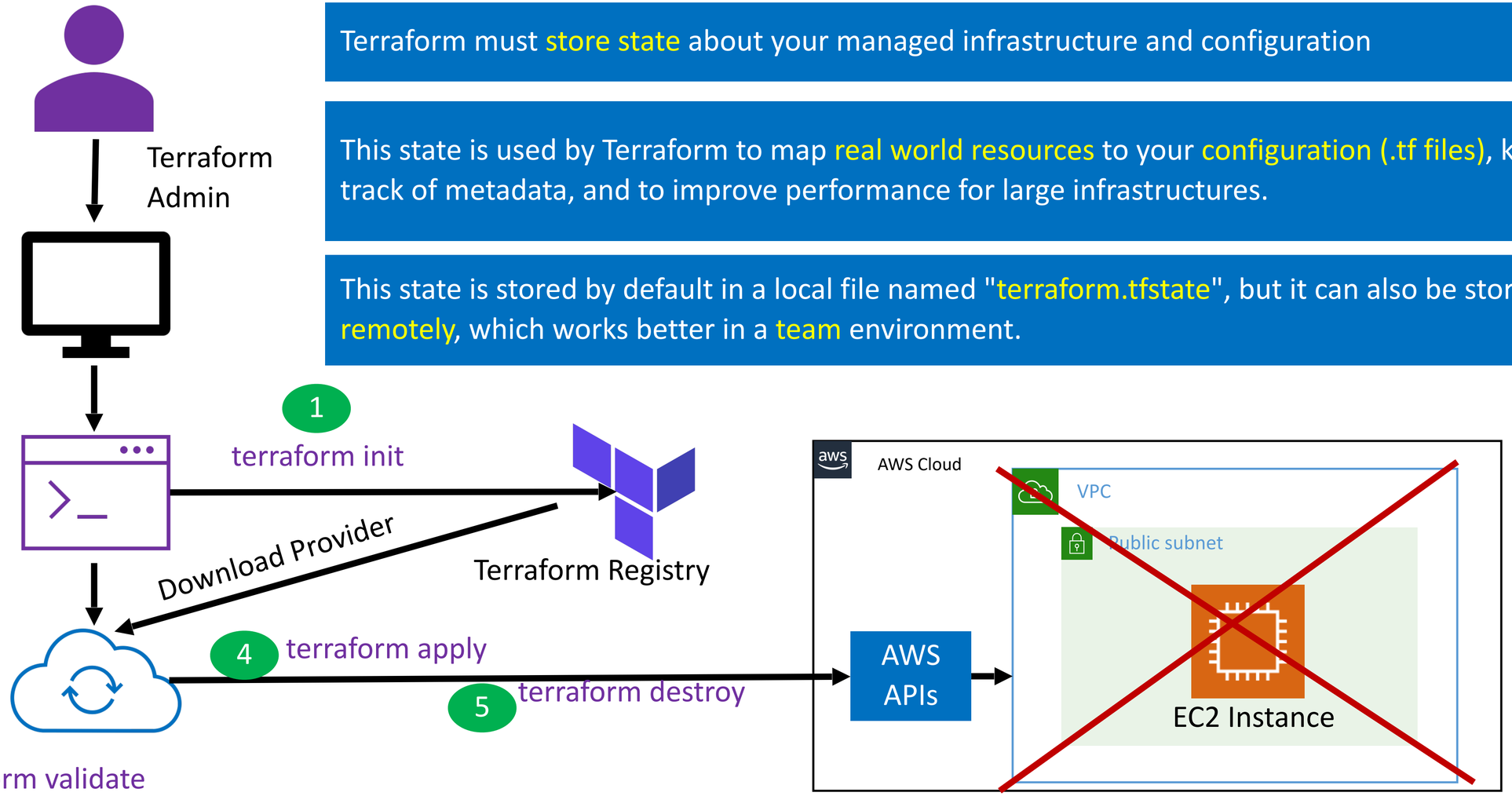
State



Resource Behavior

Terraform State

Terraform State



Terraform must store state about your managed infrastructure and configuration

This state is used by Terraform to map real world resources to your configuration (.tf files), keep track of metadata, and to improve performance for large infrastructures.

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

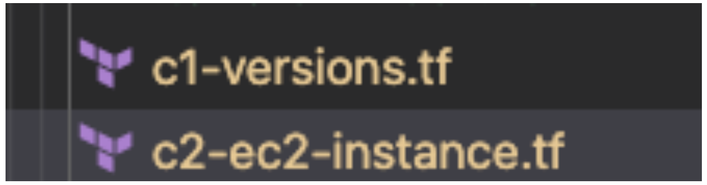
The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.

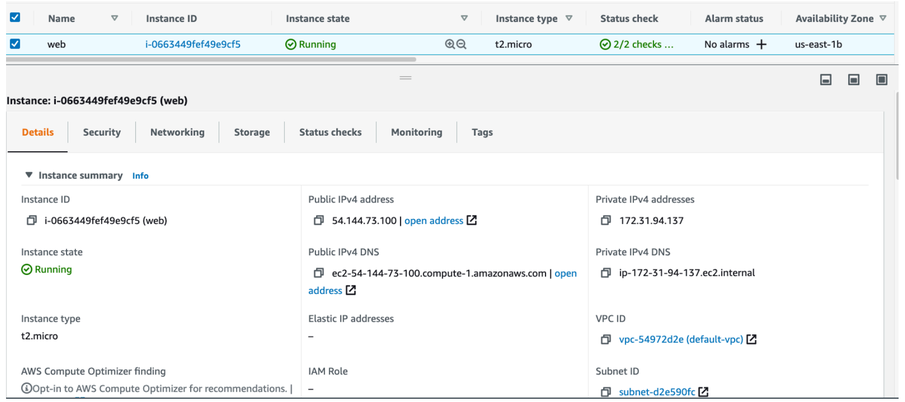
When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

Desired & Current Terraform States

Terraform Configuration Files

Real World Resource – EC2 Instance





Desired State

Current State

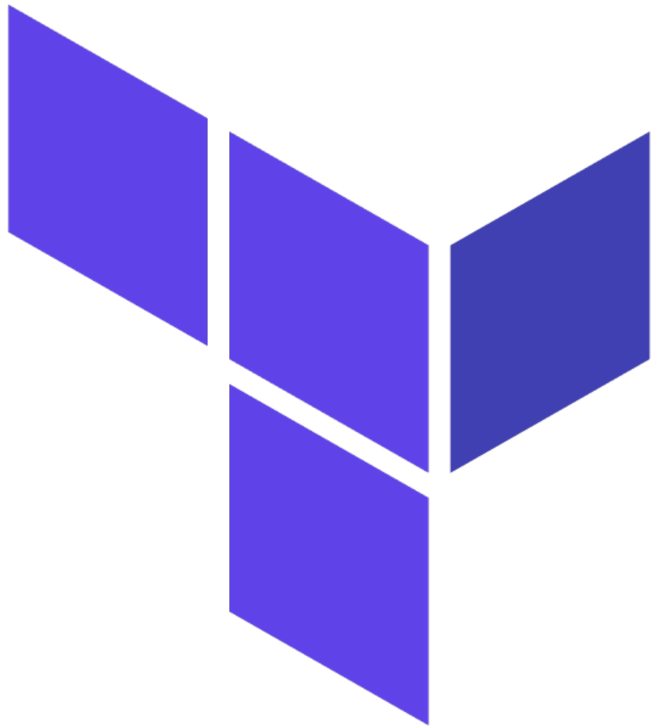
Day-3

Agenda

- Terraform State
- Terraform Datasources
- Terraform Input Variables
- Terraform Outputs
- Terraform Demo

Terraform

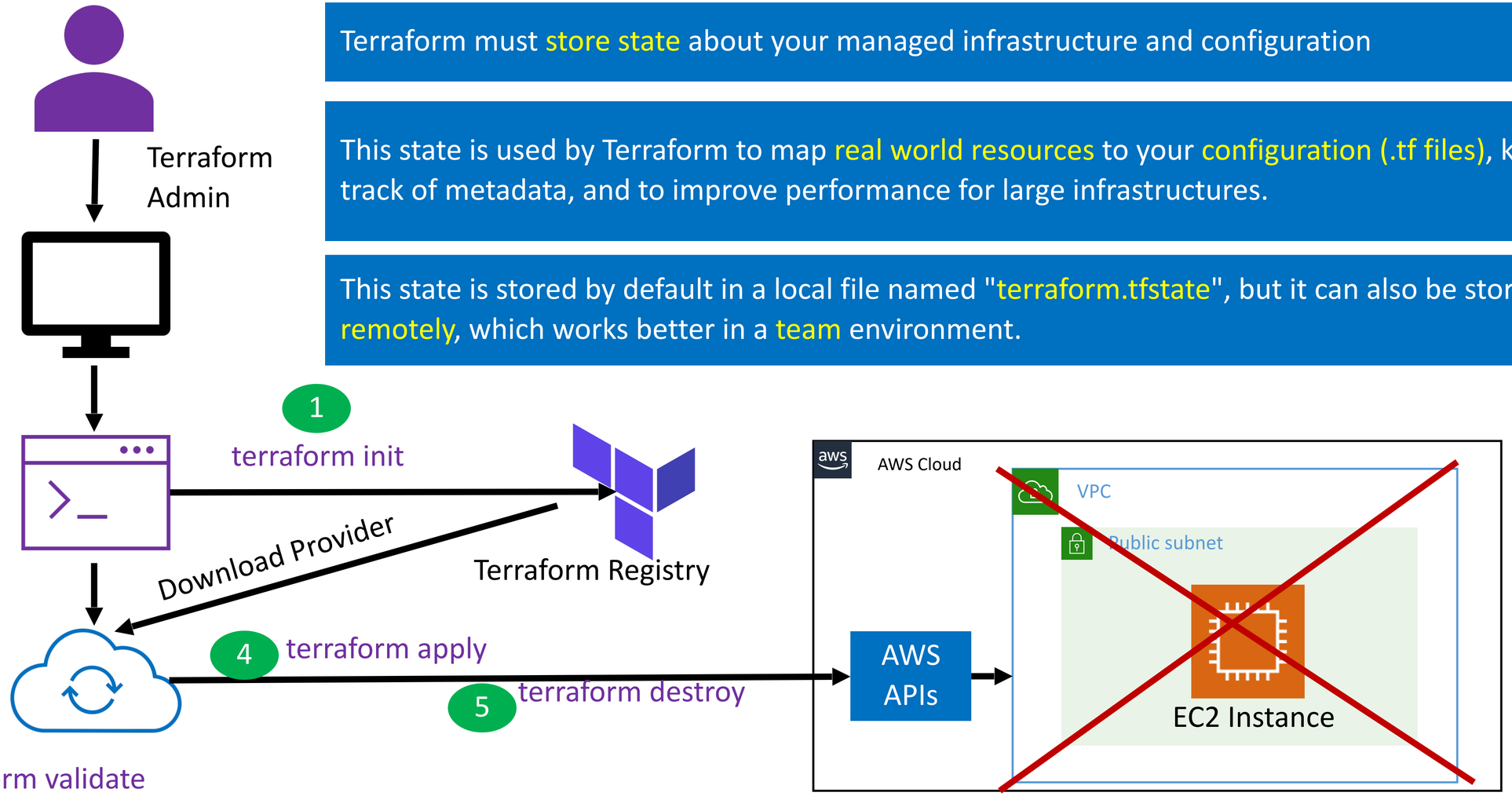
State



Resource Behavior

Terraform State

Terraform State



Terraform must store state about your managed infrastructure and configuration

This state is used by Terraform to map real world resources to your configuration (.tf files), keep track of metadata, and to improve performance for large infrastructures.

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

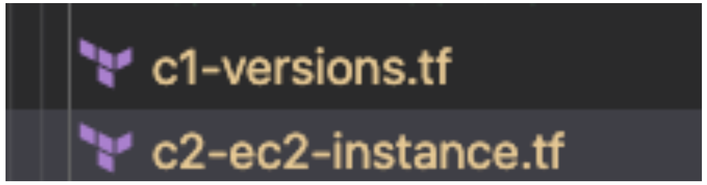
The primary purpose of Terraform state is to store bindings between objects in a remote system and resource instances declared in your configuration.

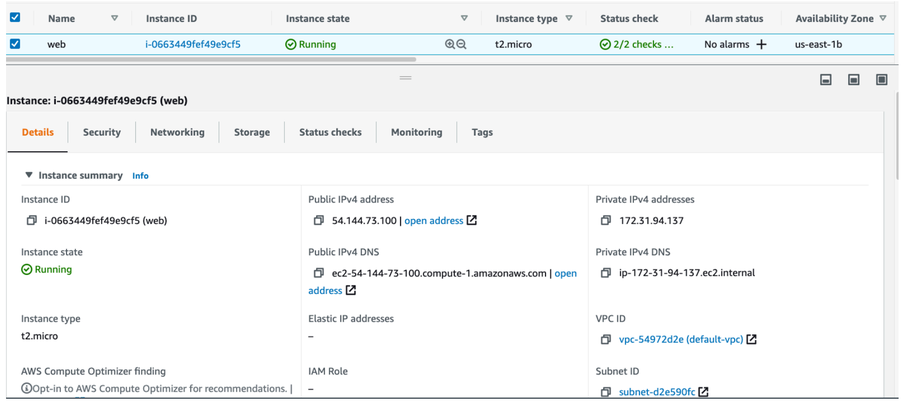
When Terraform creates a remote object in response to a change of configuration, it will record the identity of that remote object against a particular resource instance, and then potentially update or delete that object in response to future configuration changes.

Desired & Current Terraform States

Terraform Configuration Files

Real World Resource – EC2 Instance





Desired State

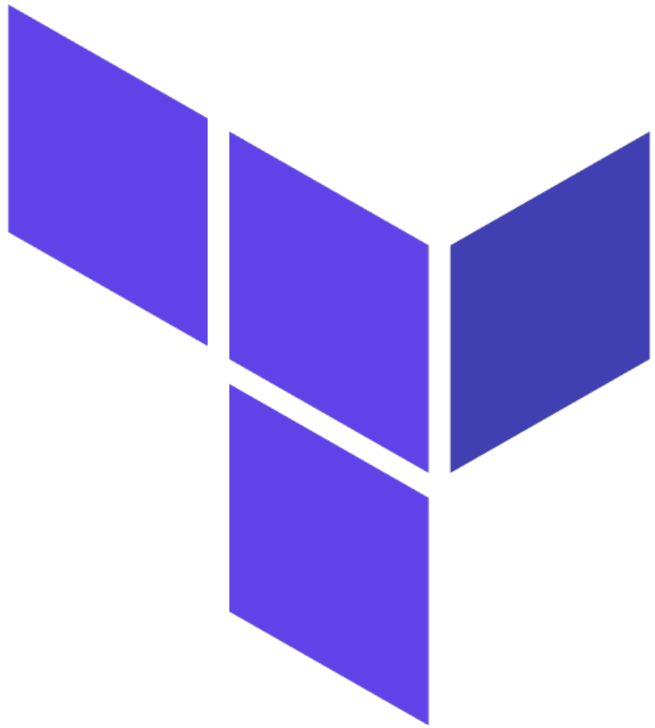
Current State

Terraform

Input Variables

Datasources

Outputs



What are we going to learn ?

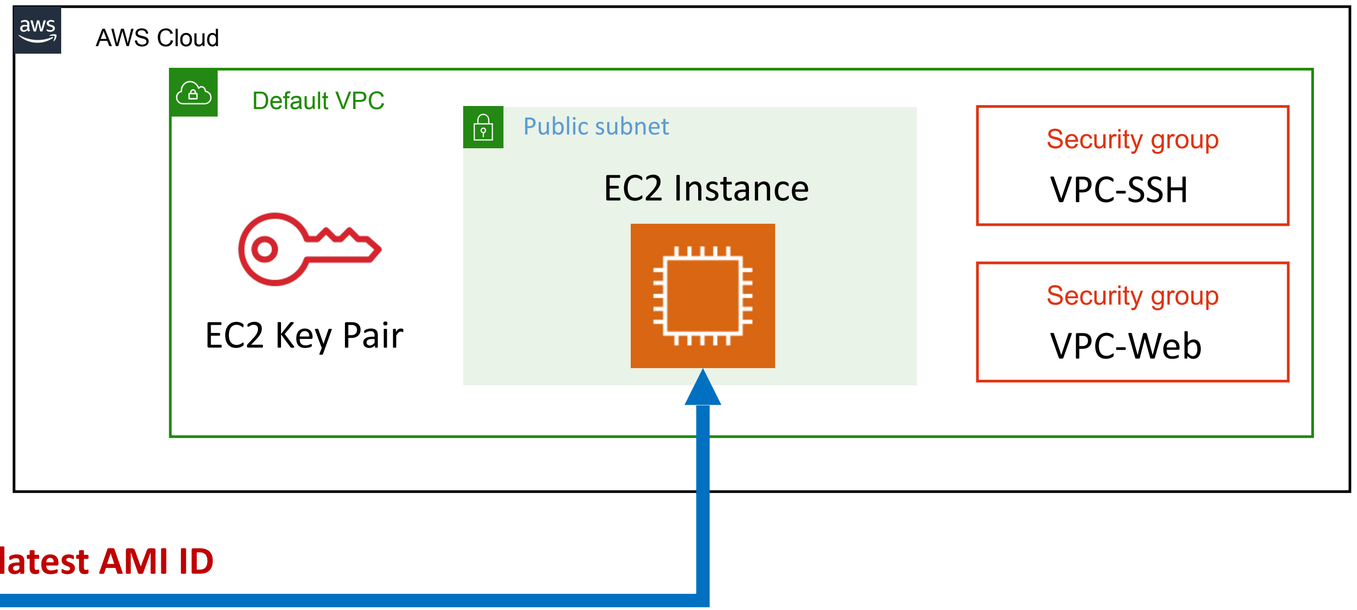
Terraform Concepts

AWS Services

Terraform Input Variables

Terraform Output Values

Terraform Datasources



Terraform Variables

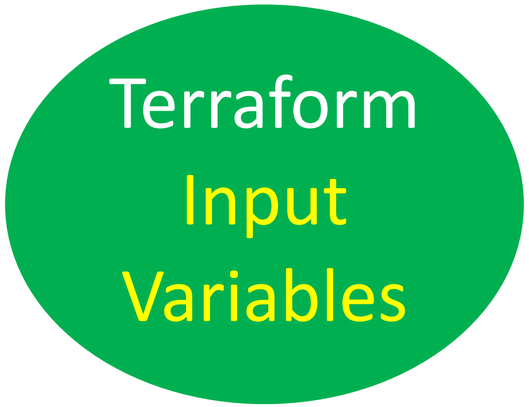
Terraform Input Variables

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 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

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

Meta-Arguments for Datasources

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

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

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

Terraform Variables – Output Values

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

Terraform Variables Outputs

1

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

2

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

Advanced

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

3

Day-4

Agenda

- Local / Remote State
- Terraform State Locking
- Terraform Backends
- Terraform State Commands
- Terraform Resource Behaviour
- Demo

Greens Technologys

Terraform State

Terraform
Local
State
Storage

Terraform
Remote
State
Storage

What is Terraform Backend ?

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

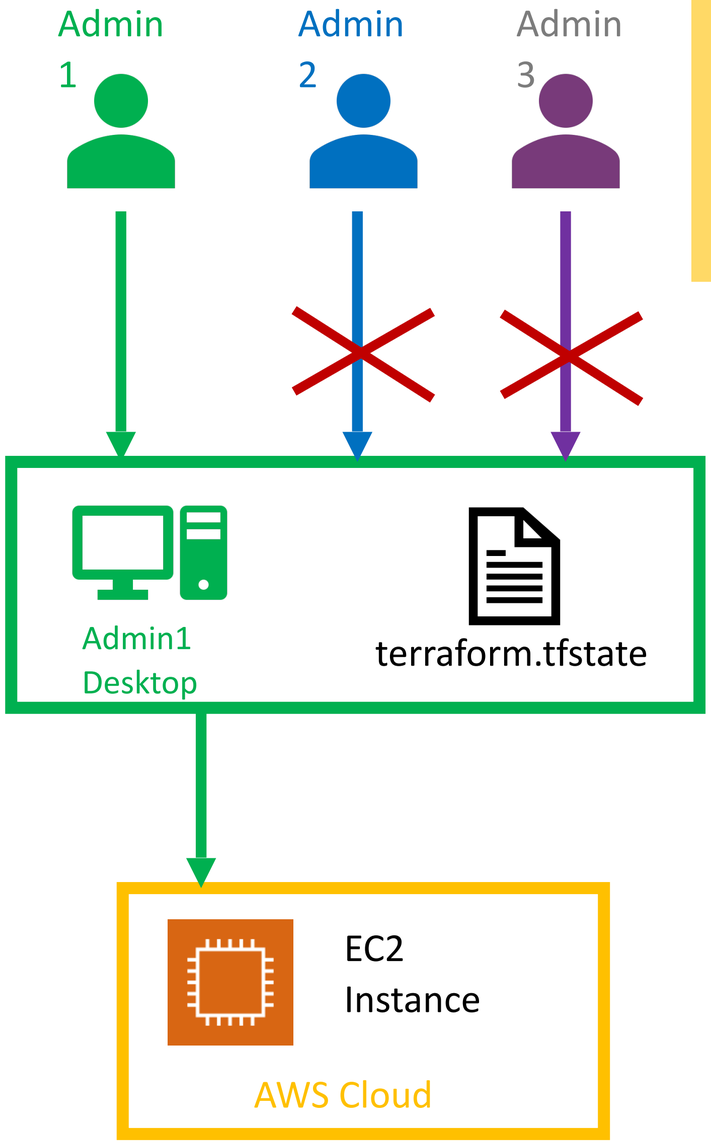
Terraform
State Storage

AWS S3 Bucket

Terraform
State Locking

AWS DynamoDB

Local State File



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

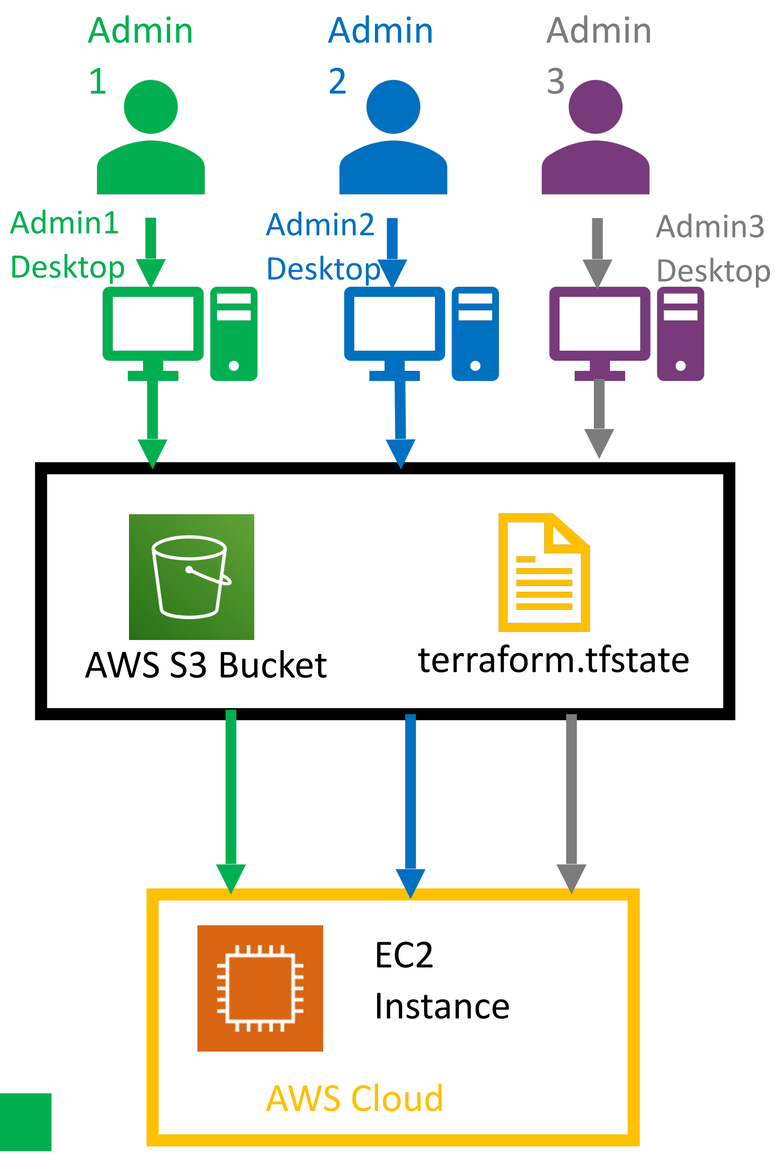
This means we need store the state file in a shared location.

Remote State File

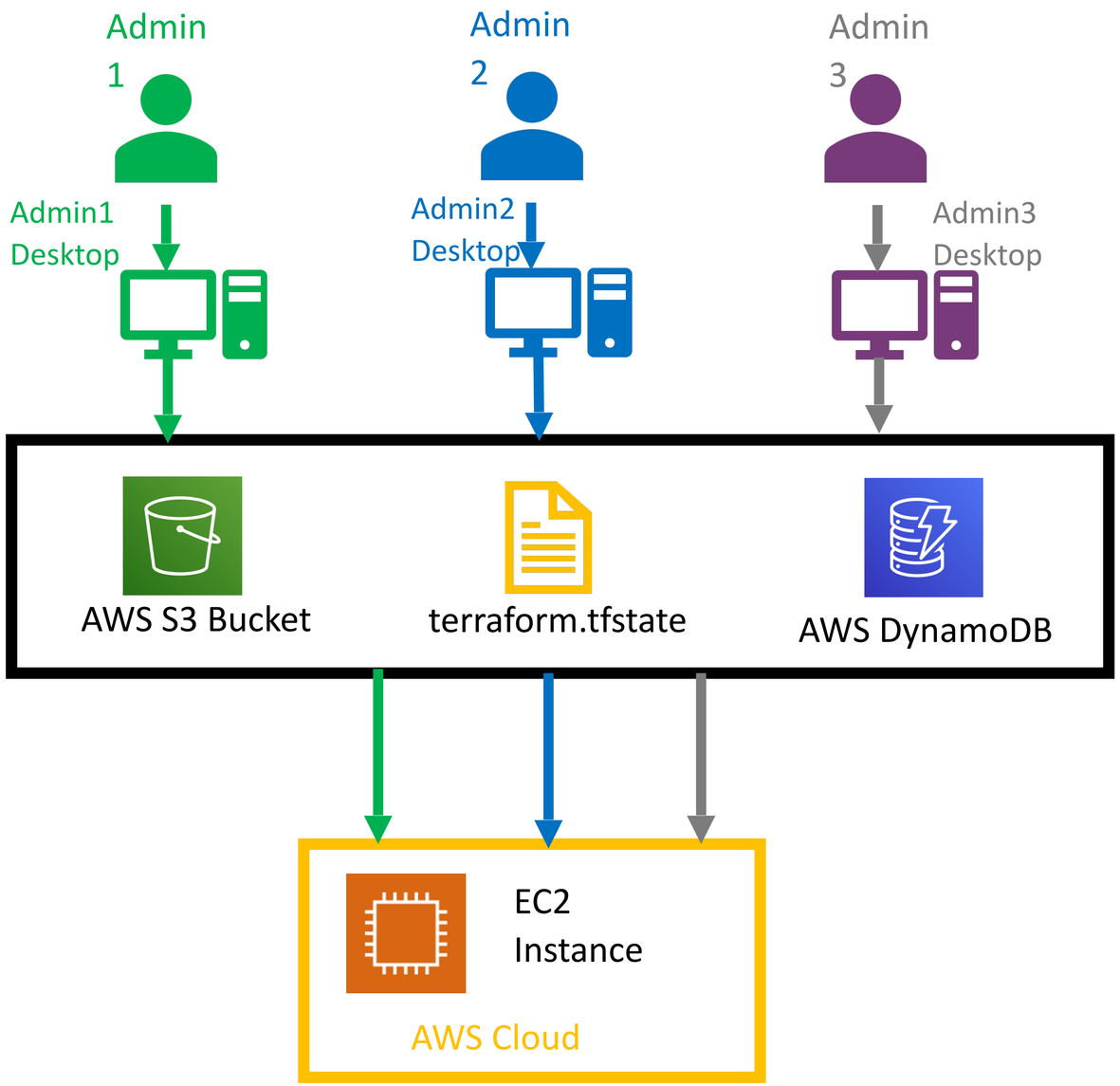
Using Terraform Backend concept we can use AWS S3 as the shared storage for State Files

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

Implement State Locking



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.

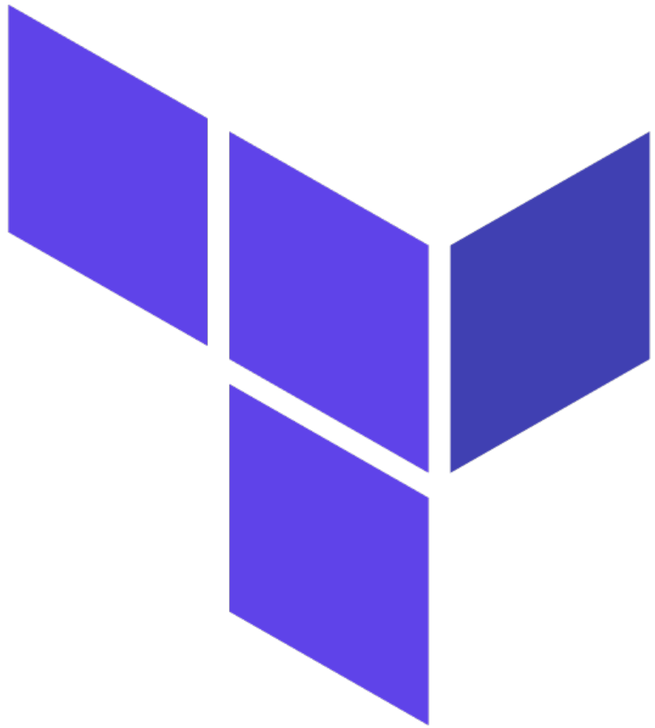
Terraform Remote State File with State Locking

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

There are two things backends will be used for

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

Store State

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

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

State Locking

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

What are Operations ?
terraform apply
terraform destroy

Operations

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

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

Only two backends actually perform operations: local and remote.

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

Terraform Backends

Backend Types

Enhanced Backends

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

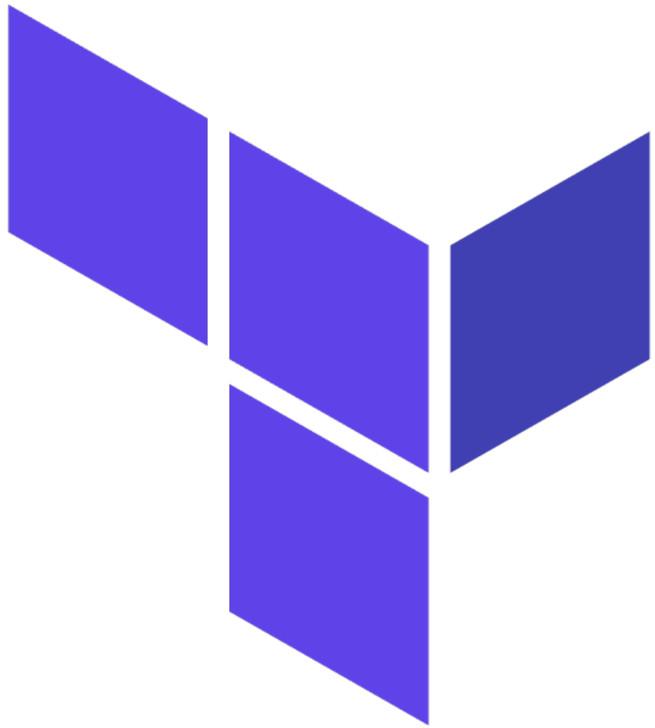
Example for Remote Backend Performing Operations : Terraform Cloud, Terraform Enterprise

Standard Backends

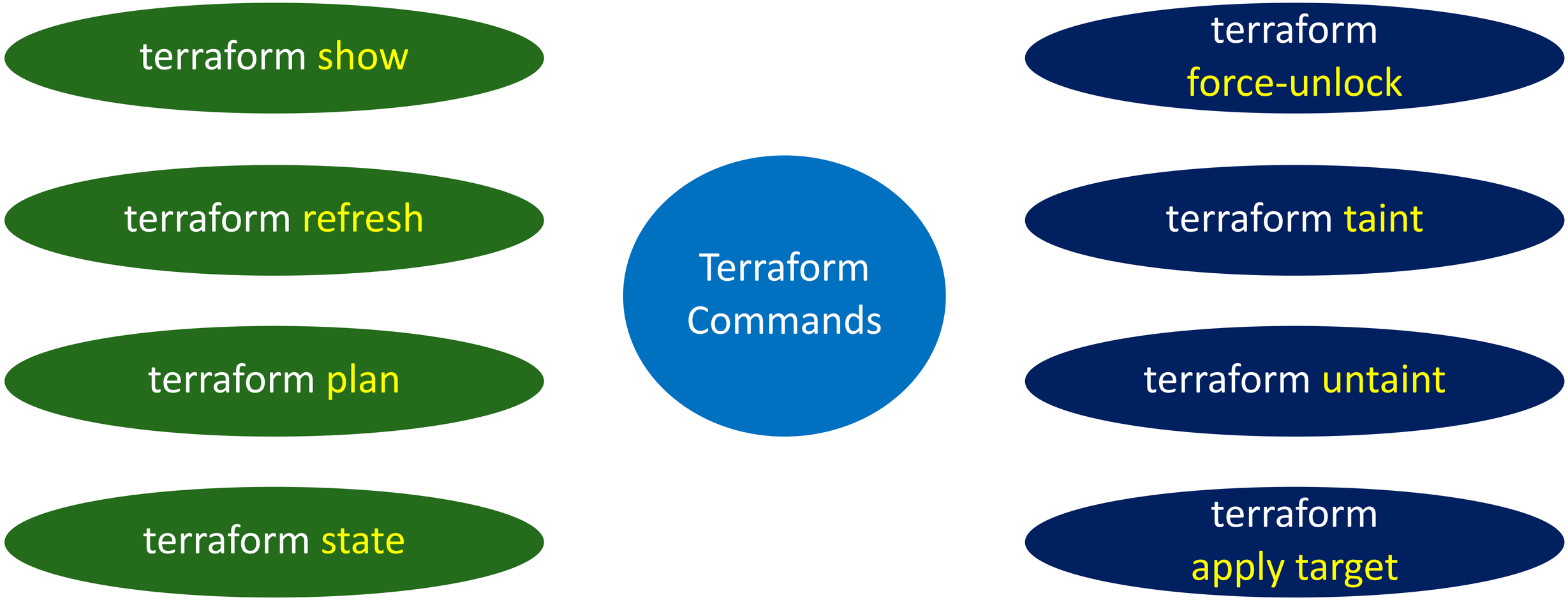
Standard backends only store state, and rely on the local backend for performing operations.

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

Terraform
State Commands



Terraform Commands – State Perspective



Day- 5

Agenda

- Terraform Modules
- Public Module Registry
- Private Modules / Version
- Terraform Demo

Greens Technologys

Build Terraform Modules Locally

Build from Scratch

If already not available, then use this approach

Review existing modules in Terraform Public Registry (Minimum 3 modules to get complete idea)

Leverage existing Terraform Modules and change the code as per your need

If already available, use the code and modify as per your need

Less effort and you can start using them in your private infrastructure spaces without public internet access

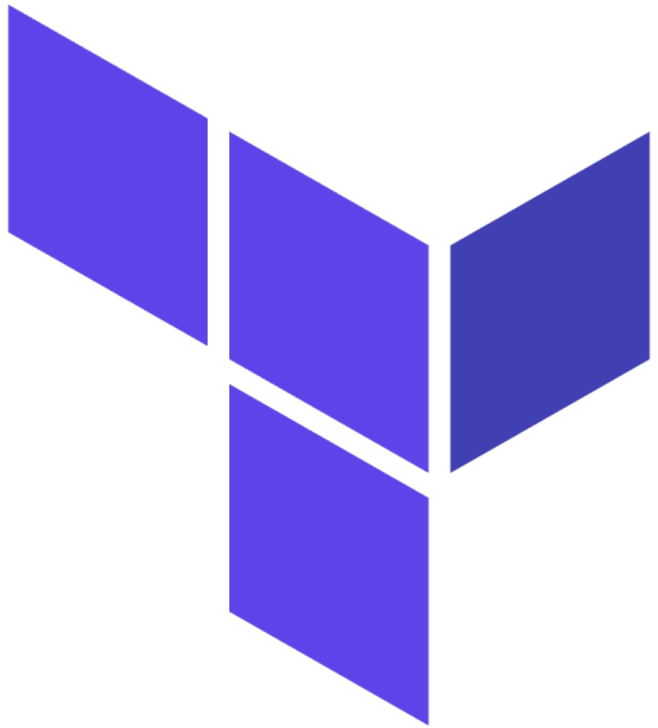
Day- 6

Agenda

- Terraform Provisioners
- File Provisioner
- local-exec Provisioner
- Remote-exec Provisioner
- Terraform Tainting / Replace
- Terraform Debugging
- Terraform Demo

Greens Technologys

Cloud



Terraform Cloud
Or
Terraform Enterprise

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

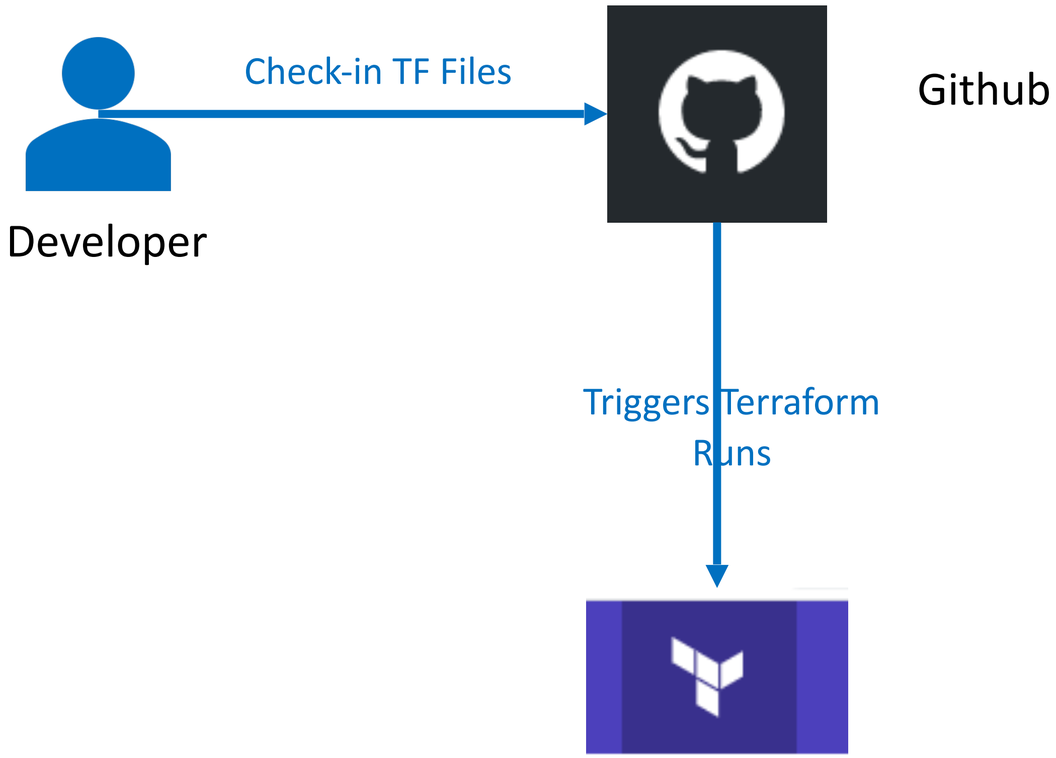
It manages Terraform runs in a consistent and reliable environment, and includes easy access to shared state and secret data, access controls for approving changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations

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

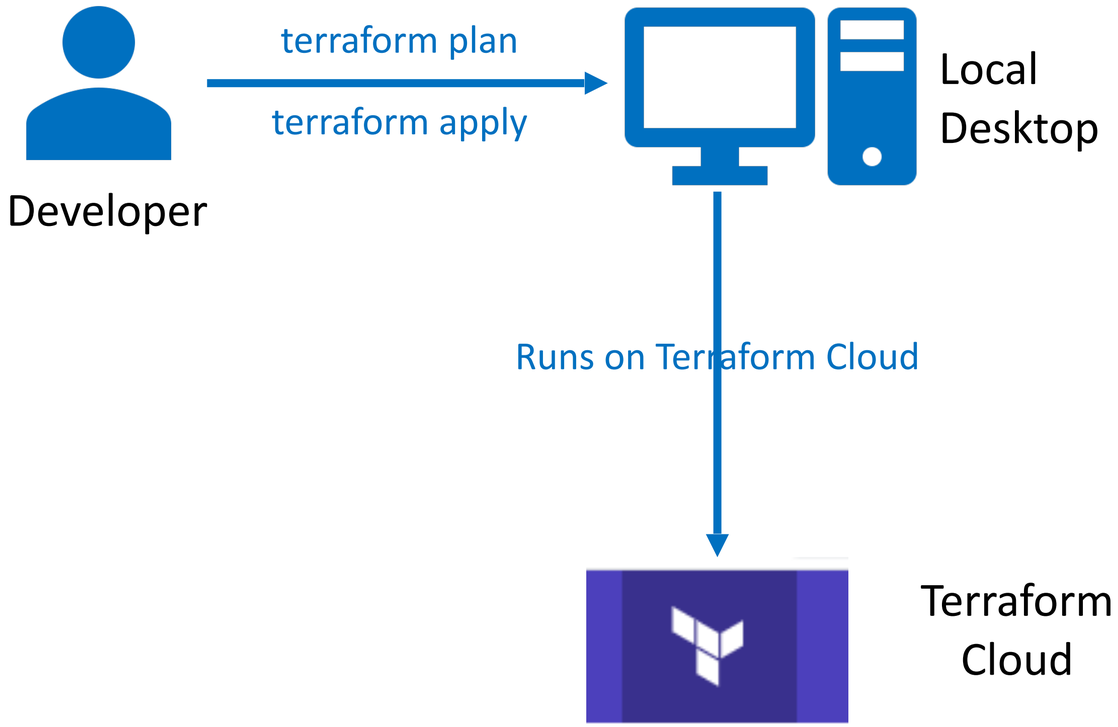
They offer free accounts for small teams, and paid plans with additional feature sets for medium-sized businesses.

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

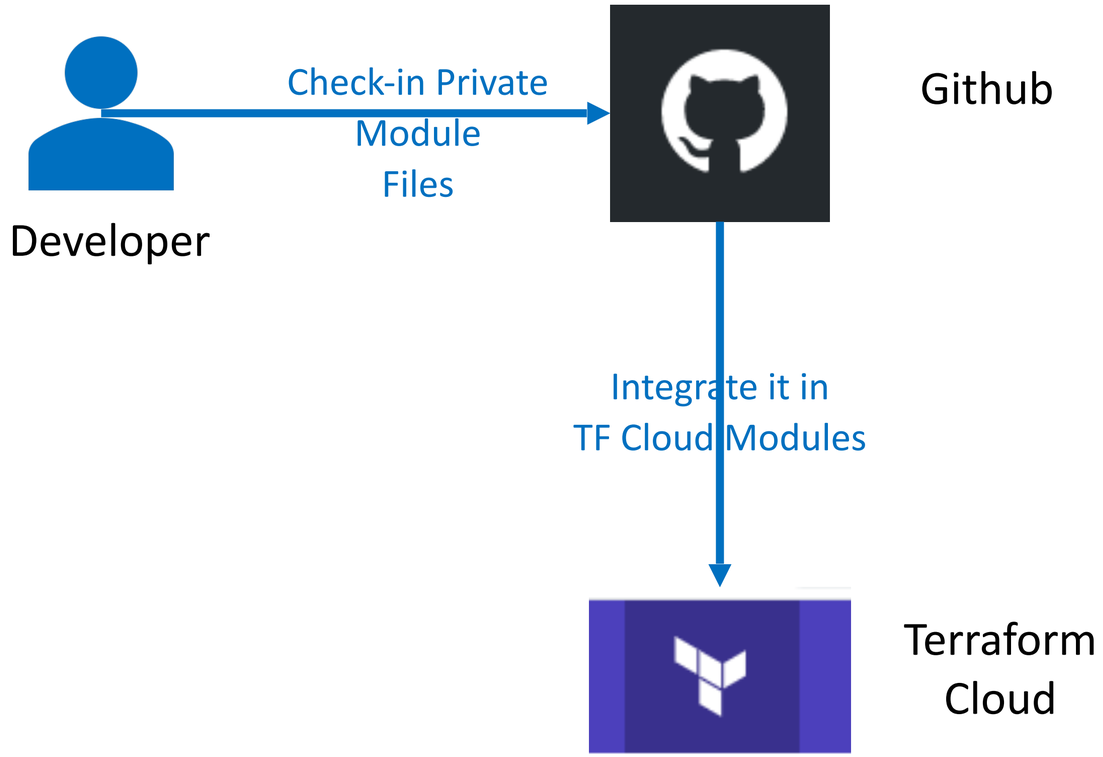
Terraform VCS-Driven Workflow



Terraform CLI-Driven Workflow



Publish Private Module Registry in Terraform Cloud



Terraform Pricing

Terraform Cloud	Free	Team & Governance	Business
Get started with our Cloud Infrastructure Automation as a Service	Collaborate on infrastructure as code for Terraform configurations.	Collaborate on infrastructure as code, manage users, and enforce provisioning policies.	Everything organizations need to use Terraform at scale.
Cloud Pricing	$0 up to 5 users	Starting at $20 user/month	Contact Sales

https://www.hashicorp.com/products/terraform/pricing

Terraform
Cloud
Version Control Workflow

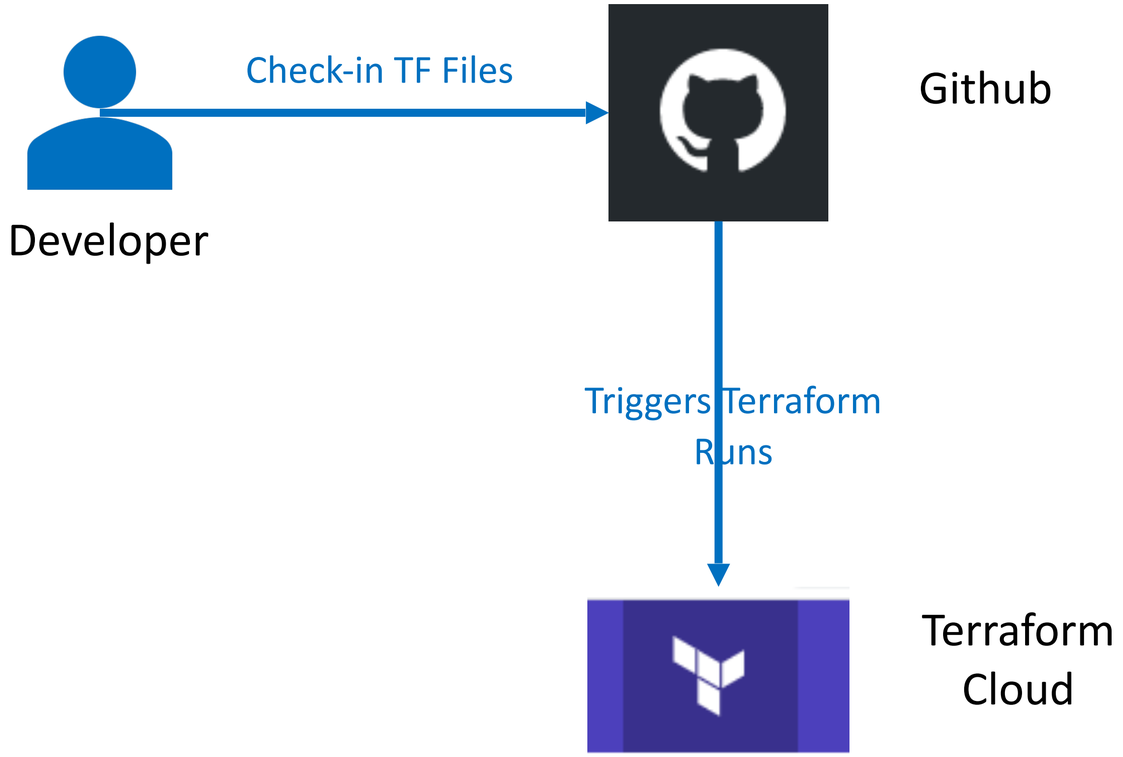
Terraform VCS Integration

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 – Supported VCS Providers

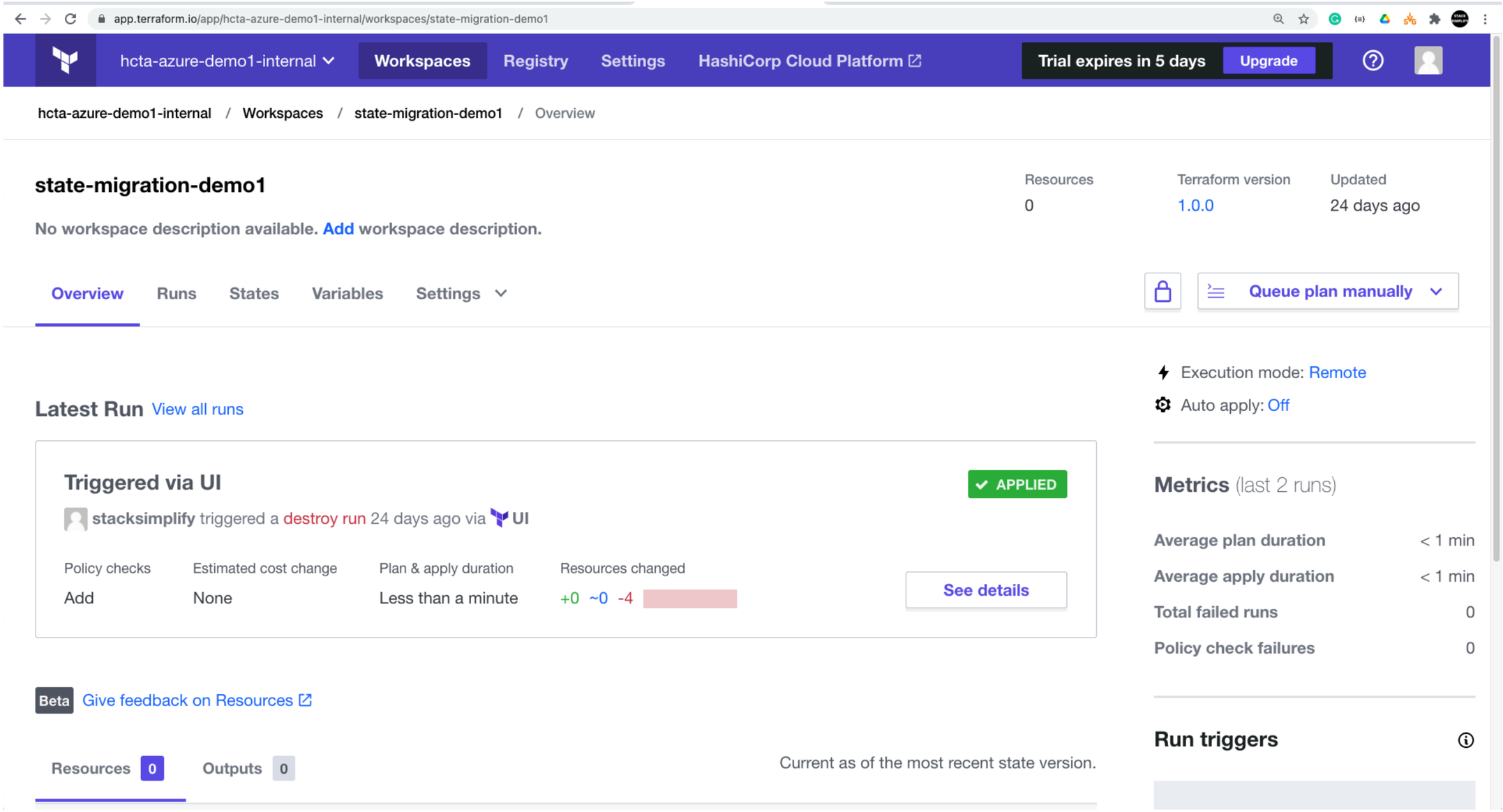
Supported VCS Providers

Terraform Cloud supports the following VCS providers:

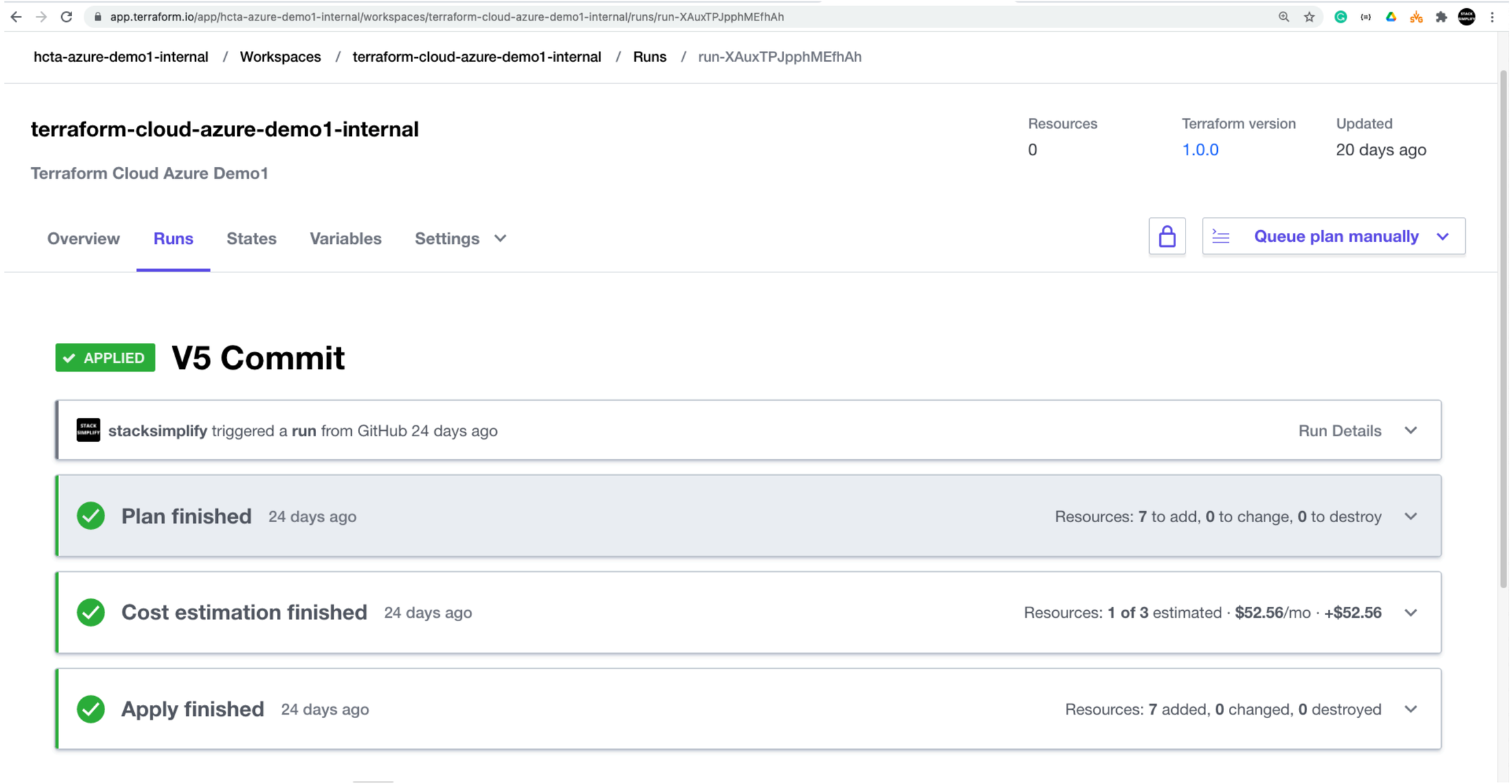
- GitHub.com
- GitHub.com (OAuth)
- GitHub Enterprise
- GitLab.com
- GitLab EE and CE
- Bitbucket Cloud
- Bitbucket Server
- Azure DevOps Server
- Azure DevOps Services

https://www.terraform.io/docs/cloud/vcs/index.html

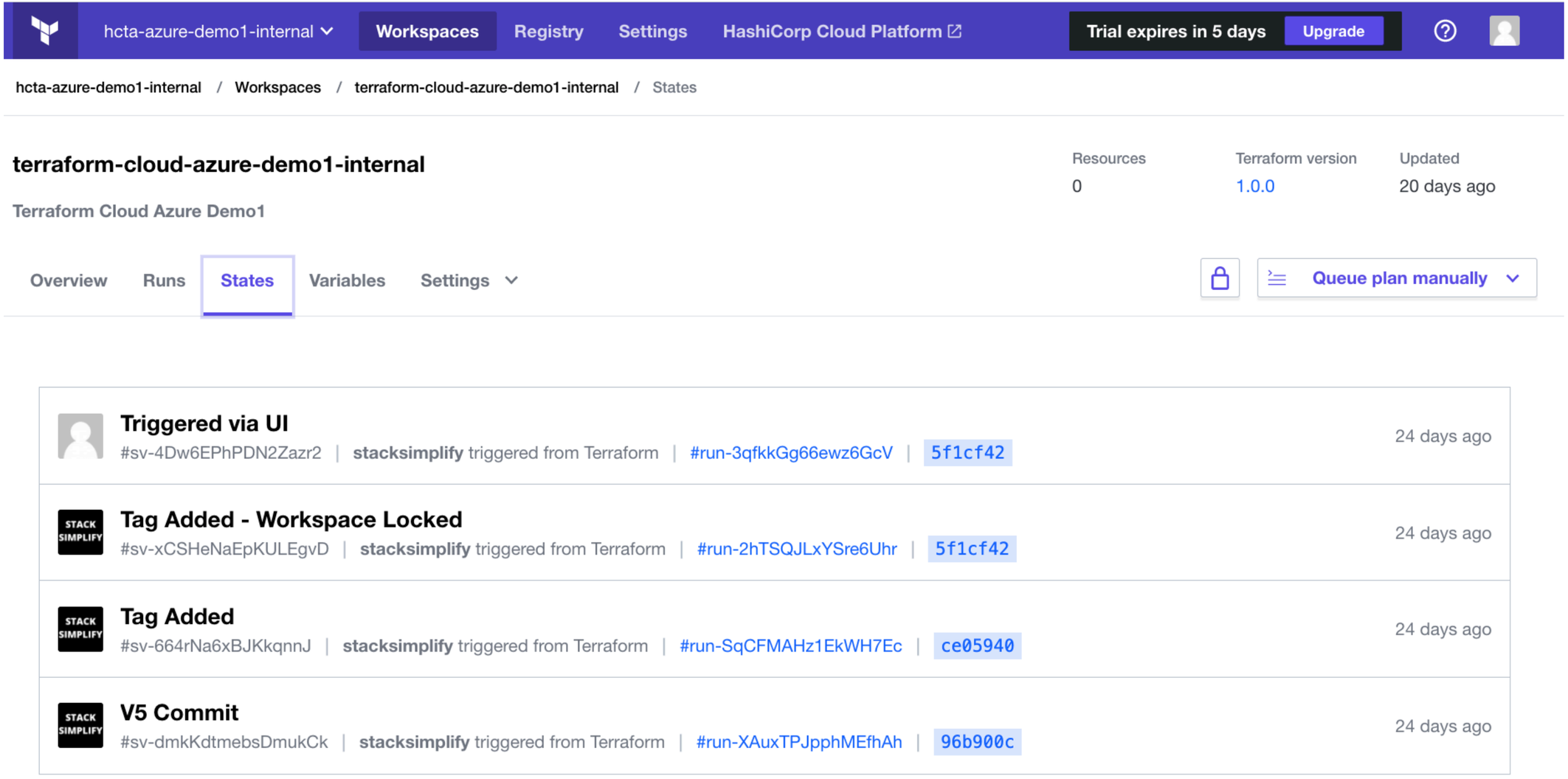
Terraform Cloud – Overview Tab



Terraform Cloud – Runs Tab

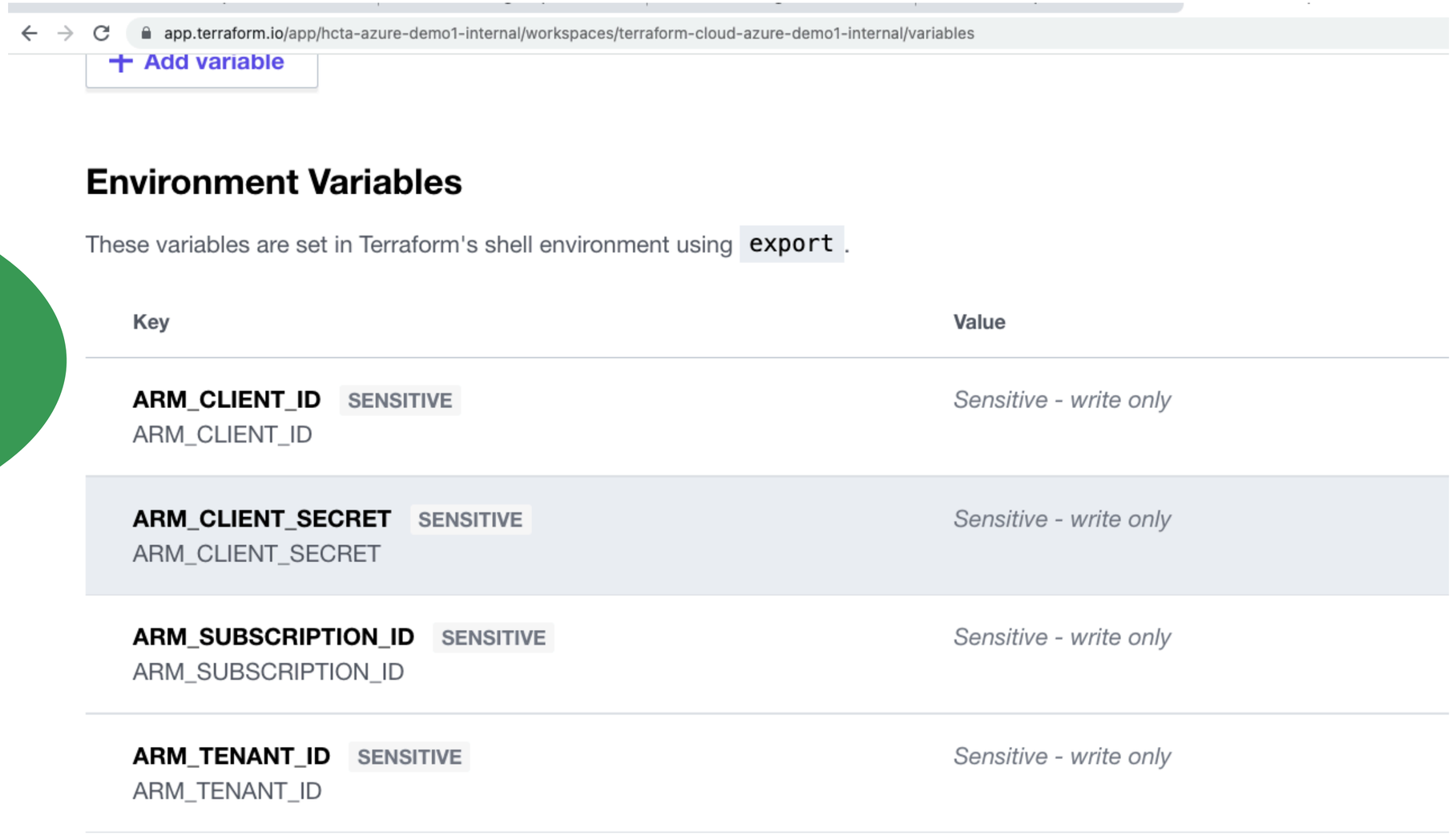


Terraform Cloud – States Tab

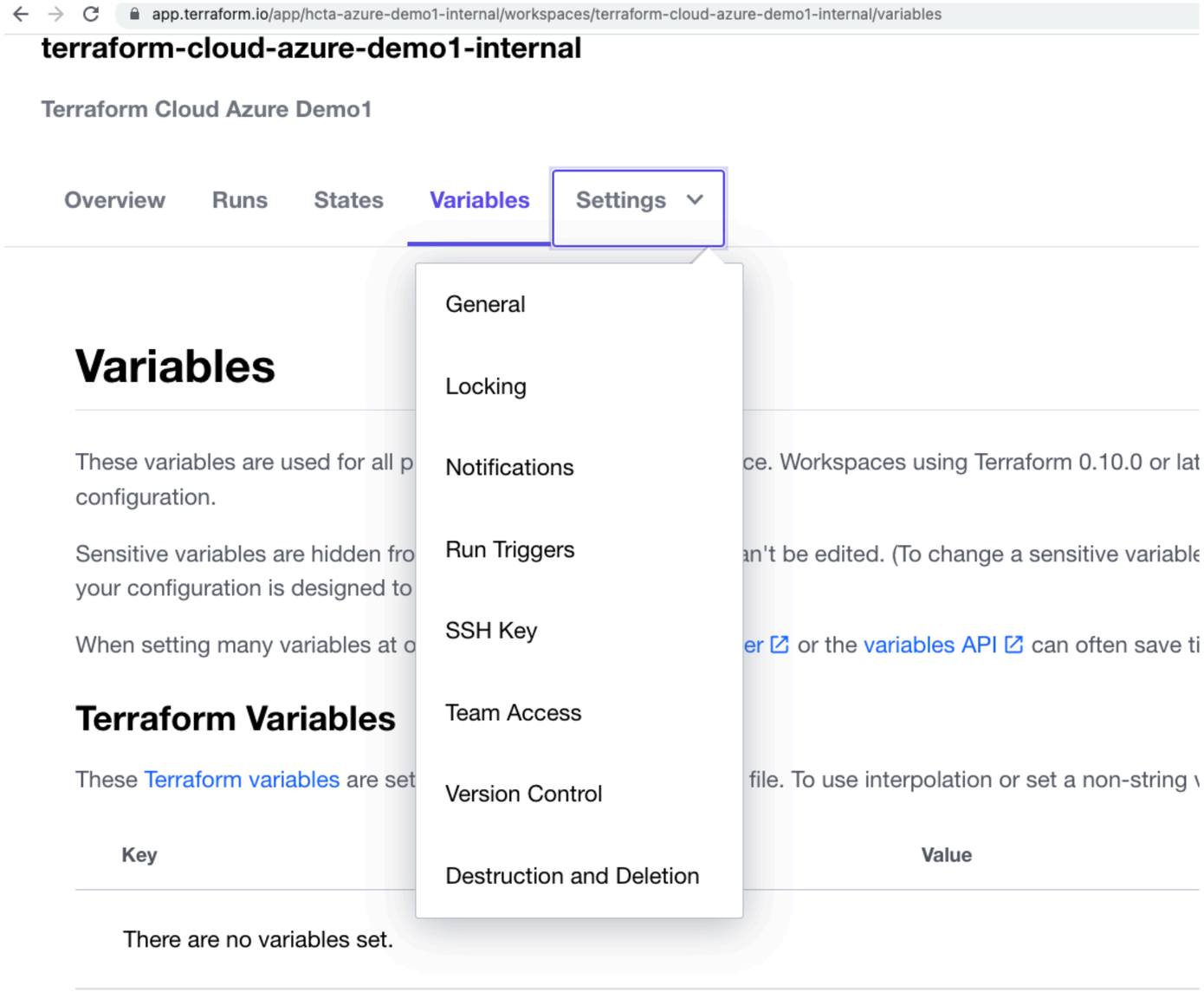


Terraform Cloud – Variables Tab

Environment Variables
defined with
Azure Client ID and Secret
to Connect to Azure Cloud



Terraform Cloud
Settings Tab



Terraform

Cloud

Private Module

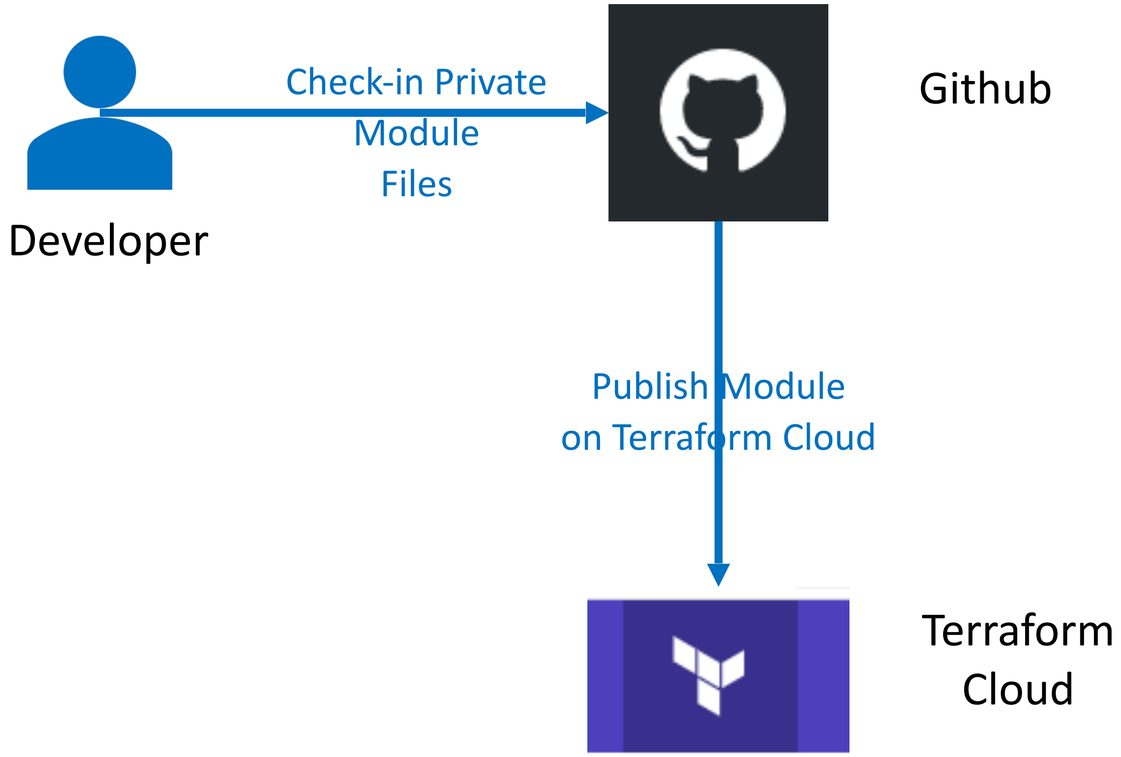
Registry

Terraform Private Module Registry

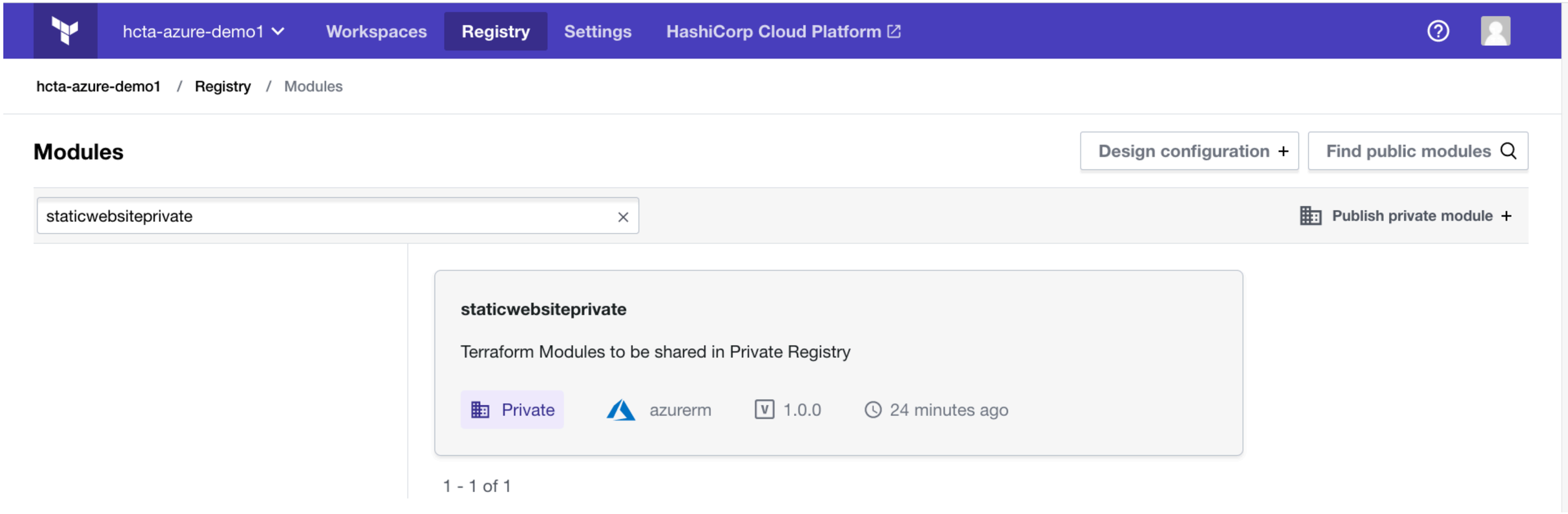
Terraform Cloud's private module registry helps you share Terraform modules across your organization.

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.

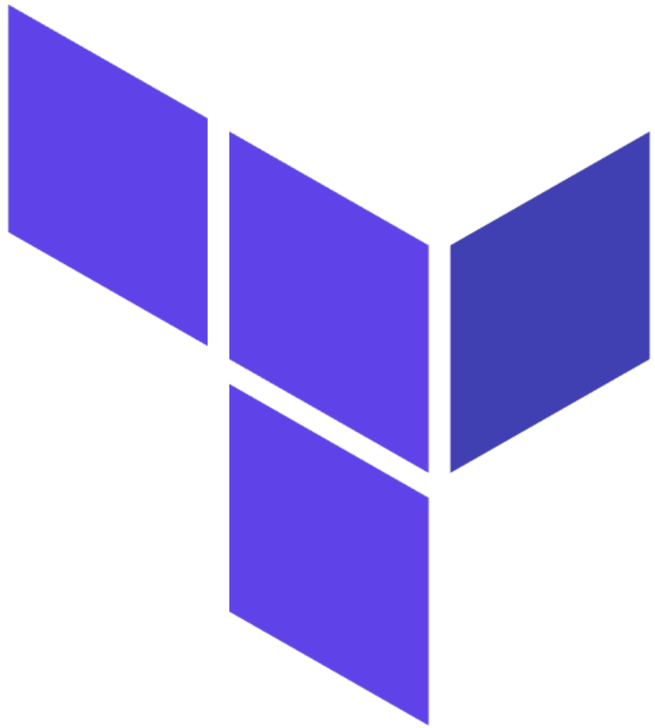
By design, the private module registry works much like the public Terraform Registry. If you're already used the public registry, Terraform Cloud's registry will feel familiar.



Terraform Cloud – Private Module Registry



Terraform
Cloud
CLI Driven Workflow



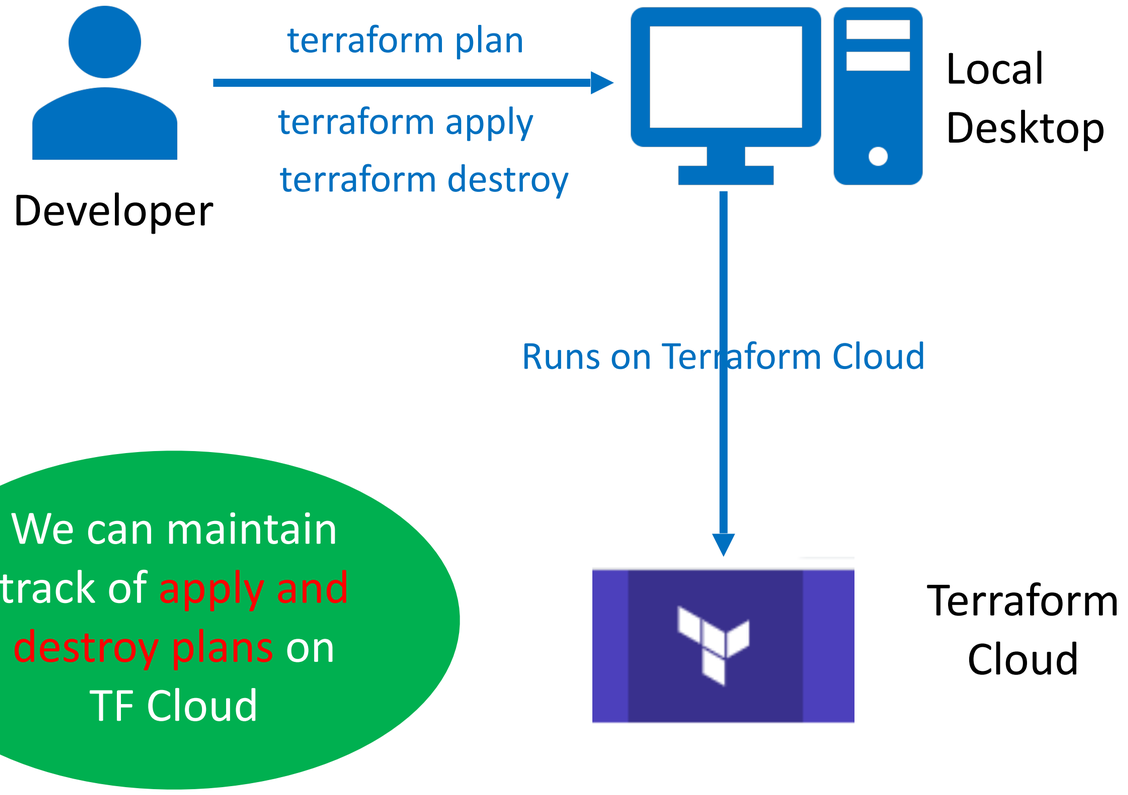
Terraform Cloud - CLI Driven Workflow

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.

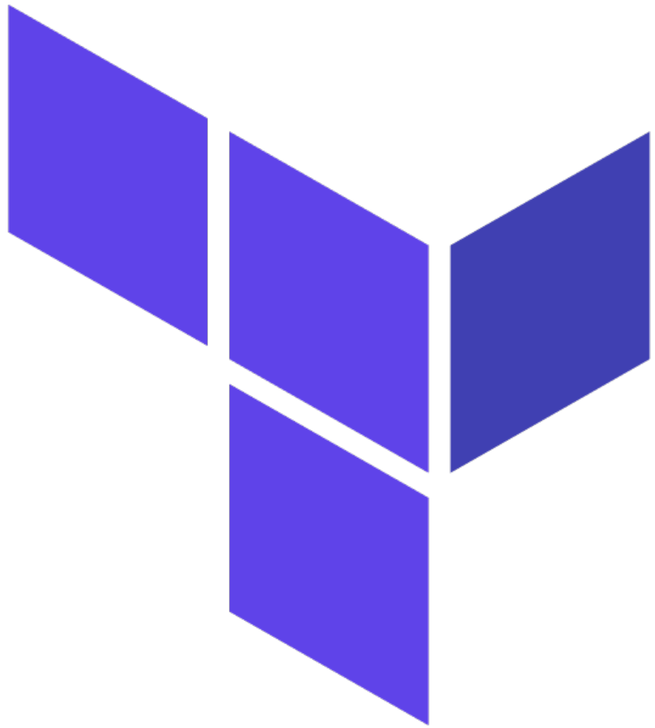
These runs execute remotely in Terraform Cloud; they use variables from the appropriate workspace, enforce any applicable Sentinel policies, and can access Terraform Cloud's private module registry and remote state inputs.



We can maintain track of apply and destroy plans on TF Cloud

Thank You

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 Policies

Plan finished a day ago

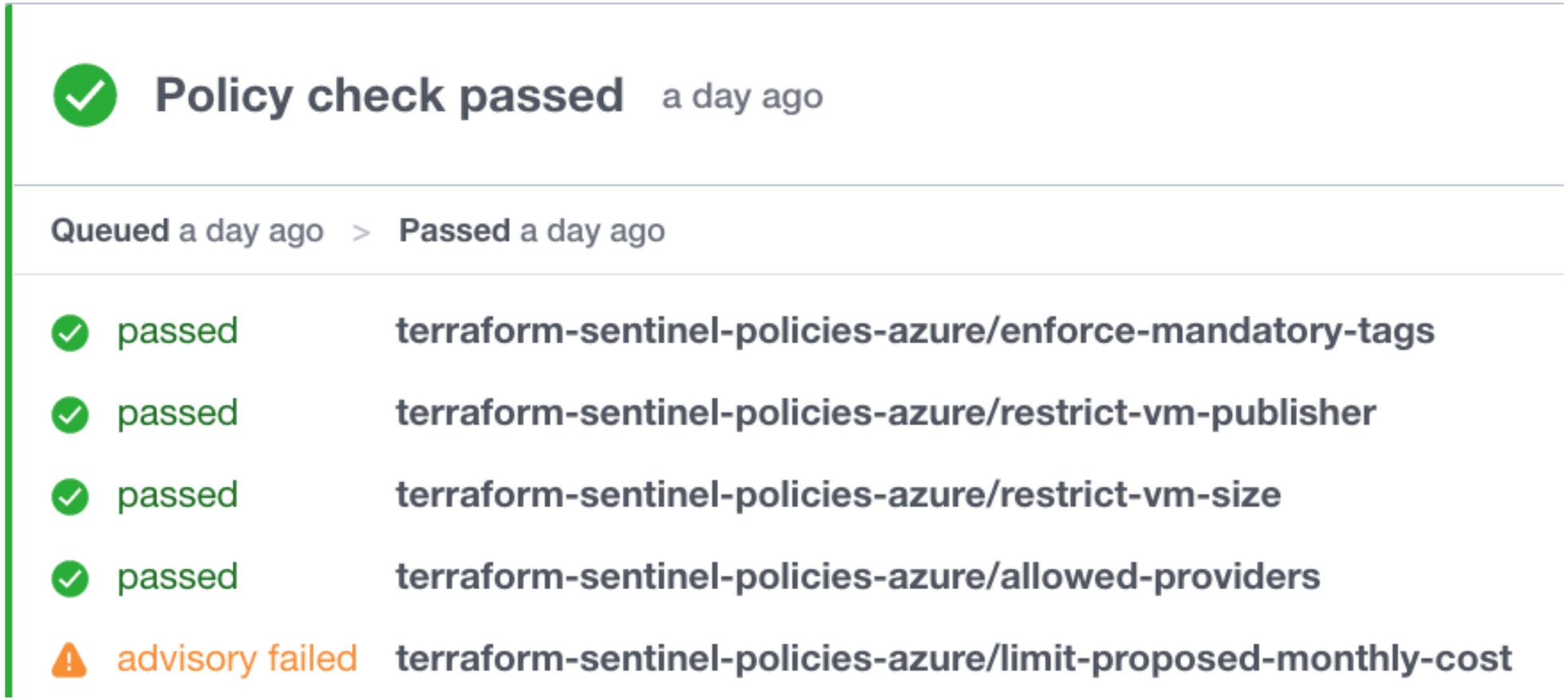
Cost estimation finished a day ago

Policy check passed a day ago

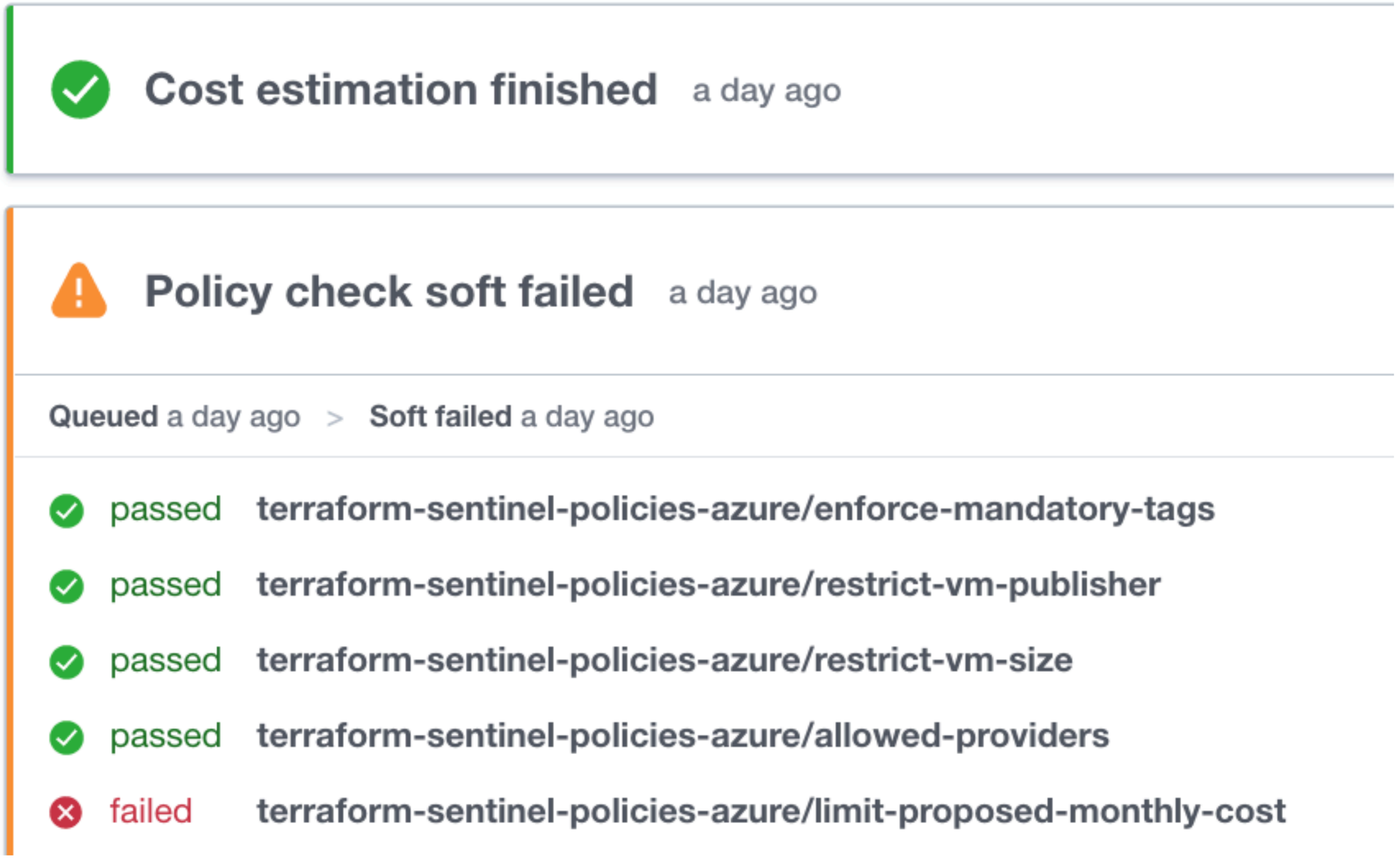
Queued a day ago > **Passed** a day ago

passed	terraform-sentinel-policies-azure/enforce-mandatory-tags
passed	terraform-sentinel-policies-azure/restrict-vm-publisher
passed	terraform-sentinel-policies-azure/restrict-vm-size
passed	terraform-sentinel-policies-azure/allowed-providers
advisory failed	terraform-sentinel-policies-azure/limit-proposed-monthly-cost

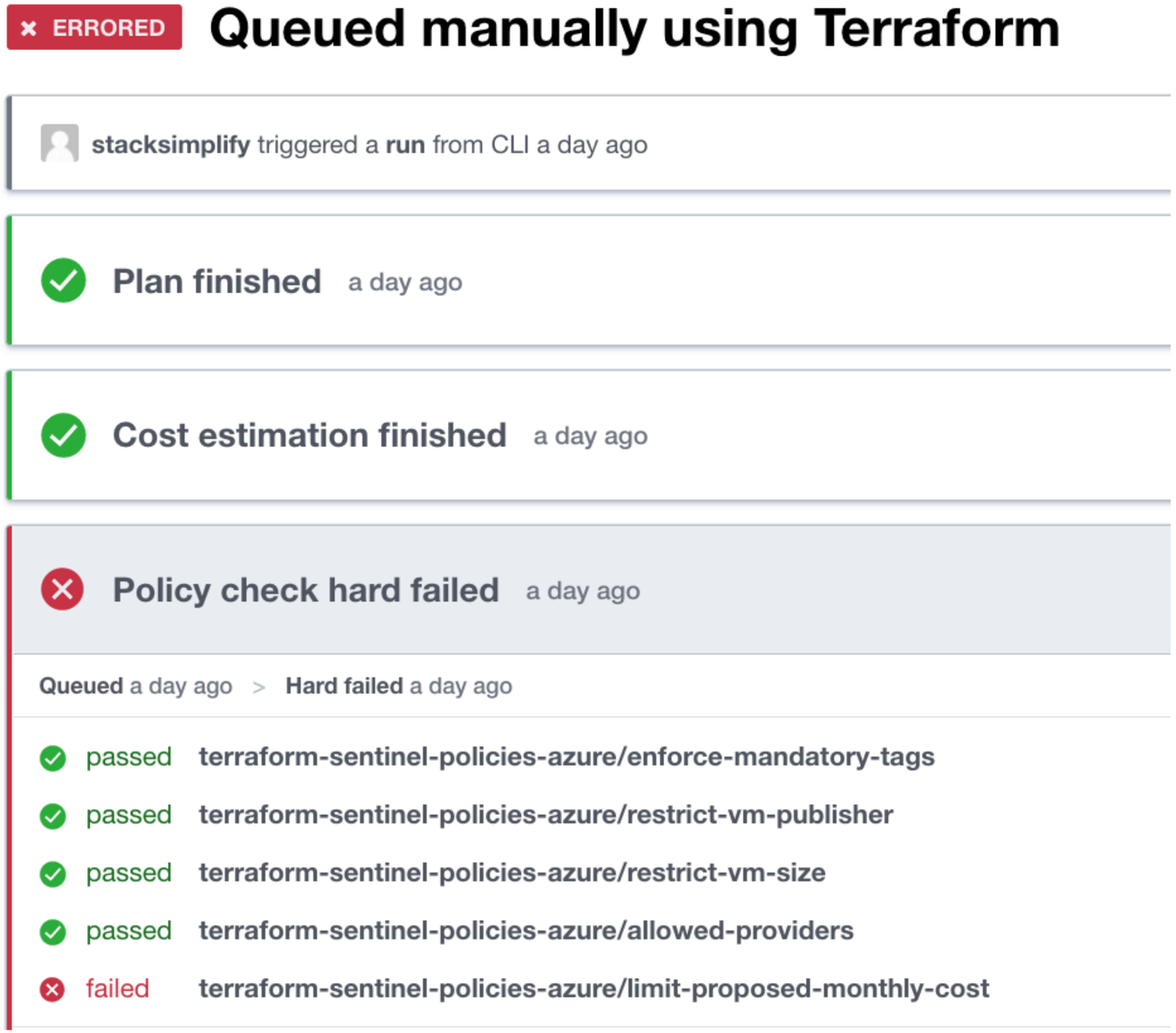
Sentinel Enforcement Mode - Advisory



Sentinel Enforcement Mode – Soft-Mandatory



Sentinel Enforcement Mode – Hard-Mandatory



Sentinel – CIS Policies

Plan finished 21 hours ago

Cost estimation finished 21 hours ago

Policy check passed 21 hours ago

Queued 21 hours ago > **Passed** 21 hours ago

passed terraform-sentinel-cis-policies/azure-cis-6.1-networking-deny-public-rdp-nsg-rules

passed terraform-sentinel-cis-policies/azure-cis-6.2-networking-deny-public-ssh-nsg-rules

passed terraform-sentinel-cis-policies/azure-cis-6.3-networking-deny-any-sql-database-ingress

passed terraform-sentinel-cis-policies/azure-cis-6.4-networking-enforce-network-watcher-flow-log-retention-period