# Terraform – Training - AWS

# Agenda

**Introduction**

- Challenges and overview of IAC
- Architecture
- Directory structure
- Terraform workflow
- Terraform commands basics
- Terraform language basics
- Terraform state files

**Security (Authentication and Authorization)**

- The right way to handle Access & Secret keys
- Terraform and Identity Access Management (IAM)

# Agenda..Continued..

**Labs**

- Terraform installation
- Aws cli installation
- Authentication and authorization using IAM keys

**Fundamental Blocks**

- terraform
- provider
- Resource

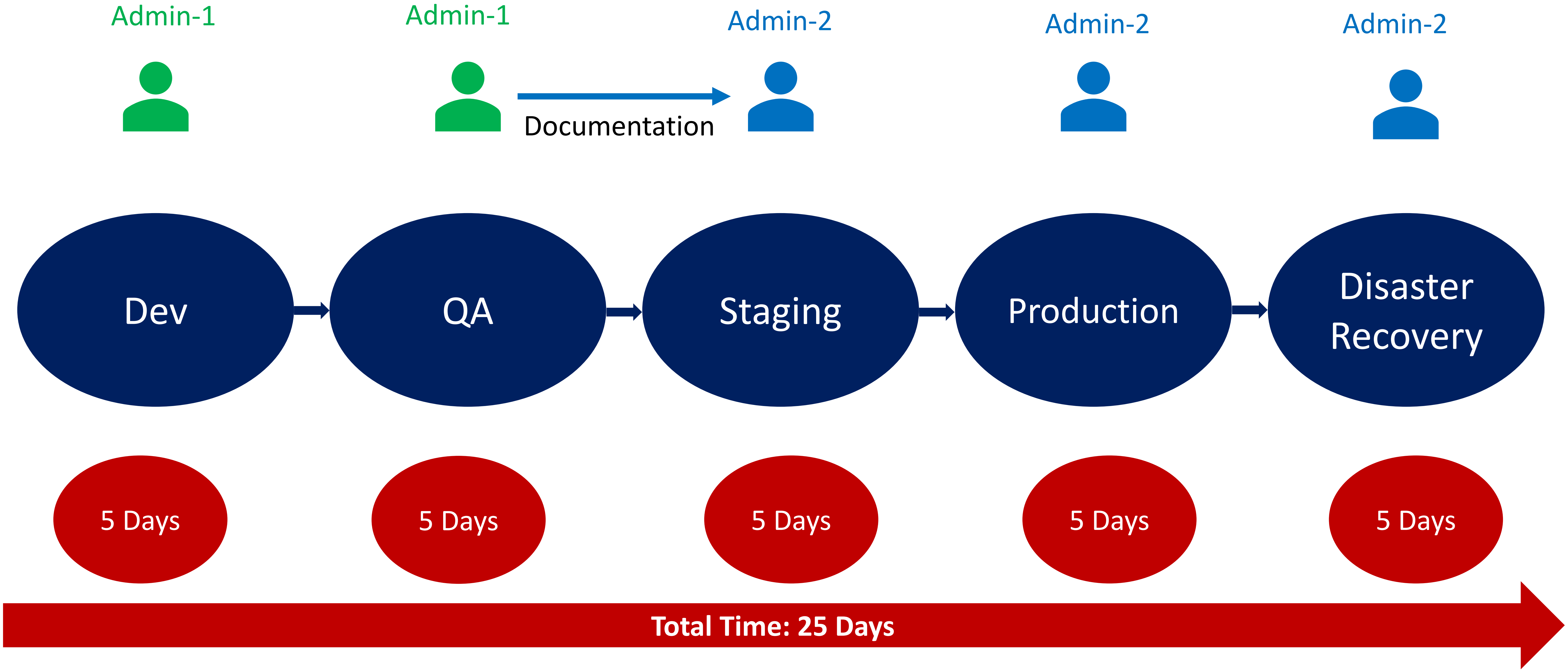
# Additional Components

**Enhancement  Blocks**

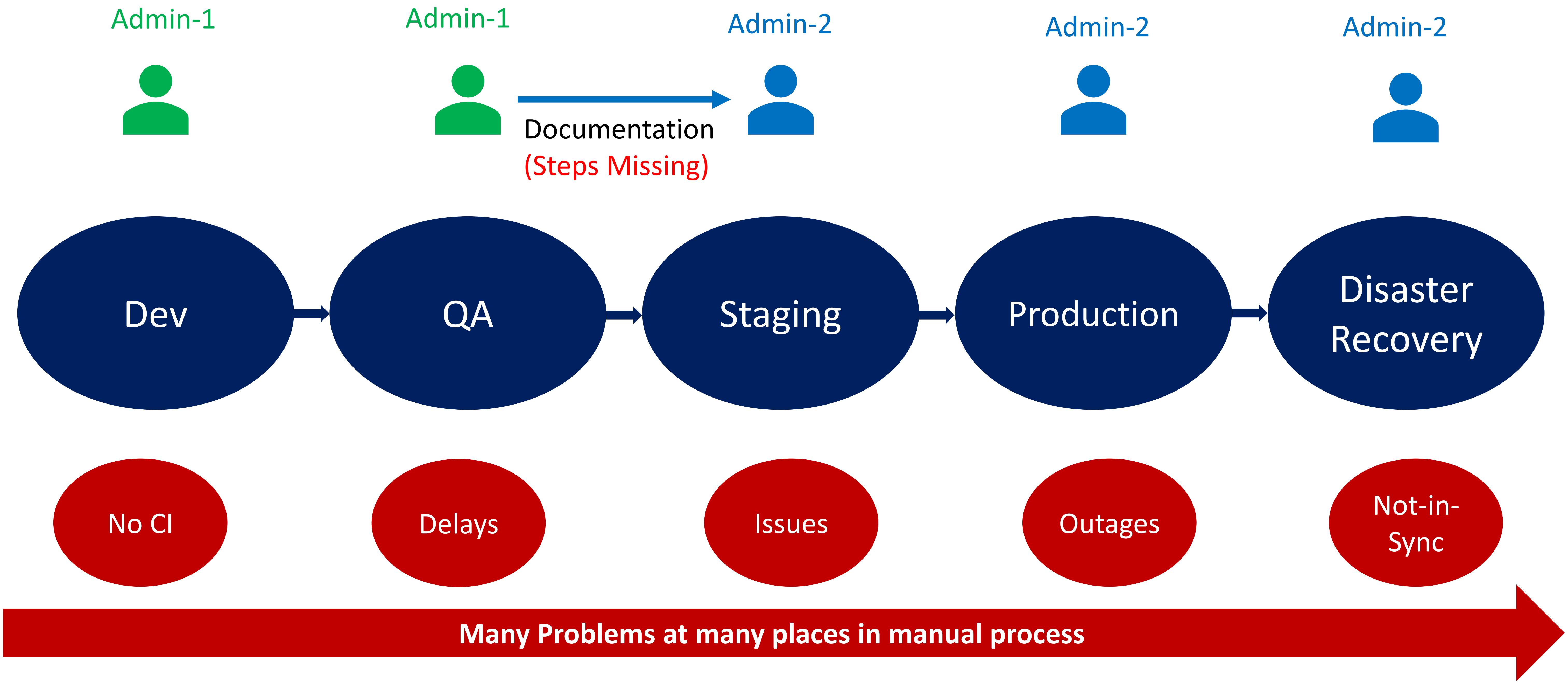
- variables
- datasource
- Metadata
- Backend (remote state management)
- Provisioners
- Modules

# Challenges Before Terraform

# Traditional Way of Managing Infrastructure

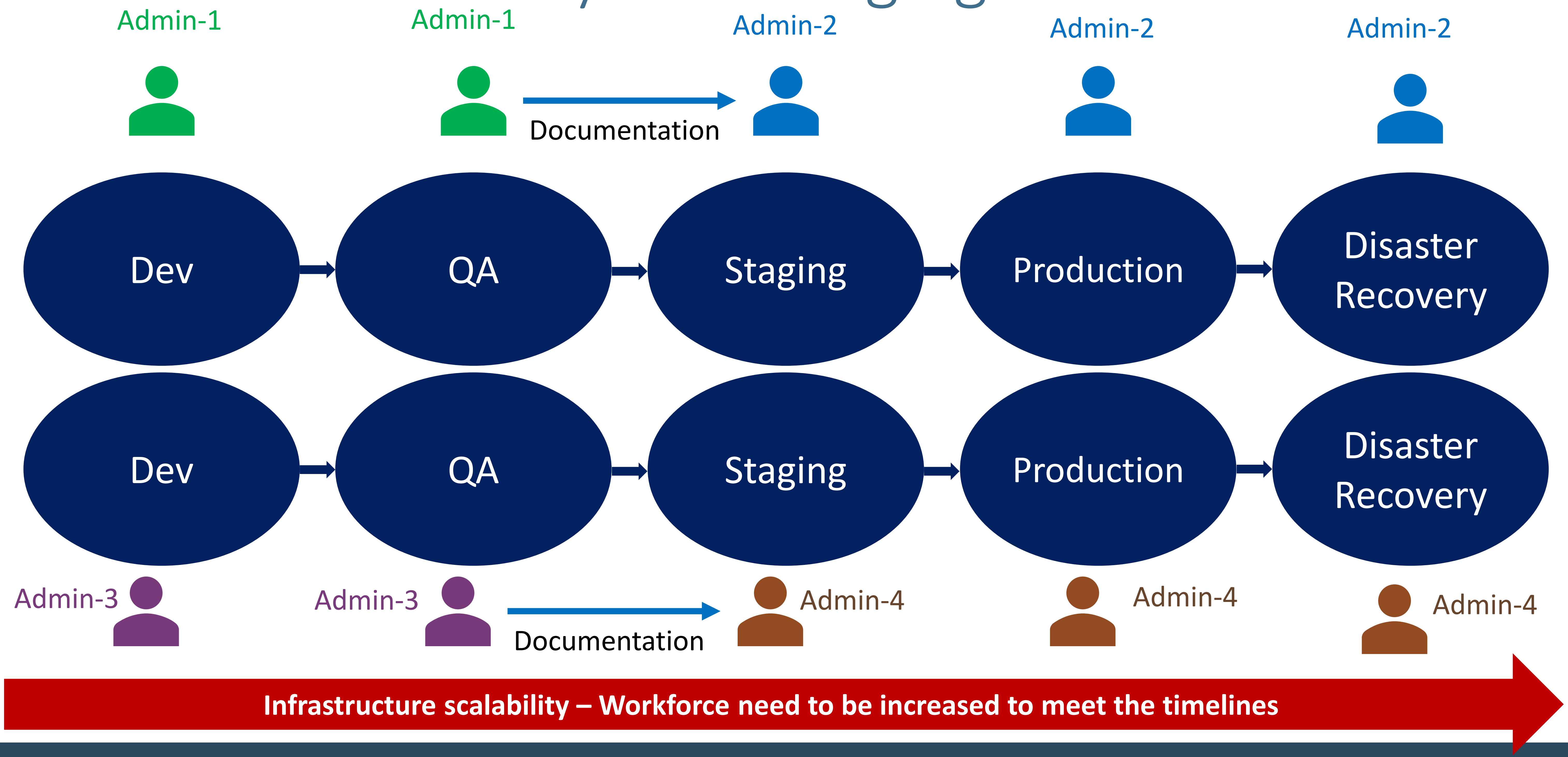


# Traditional Way of Managing Infrastructure



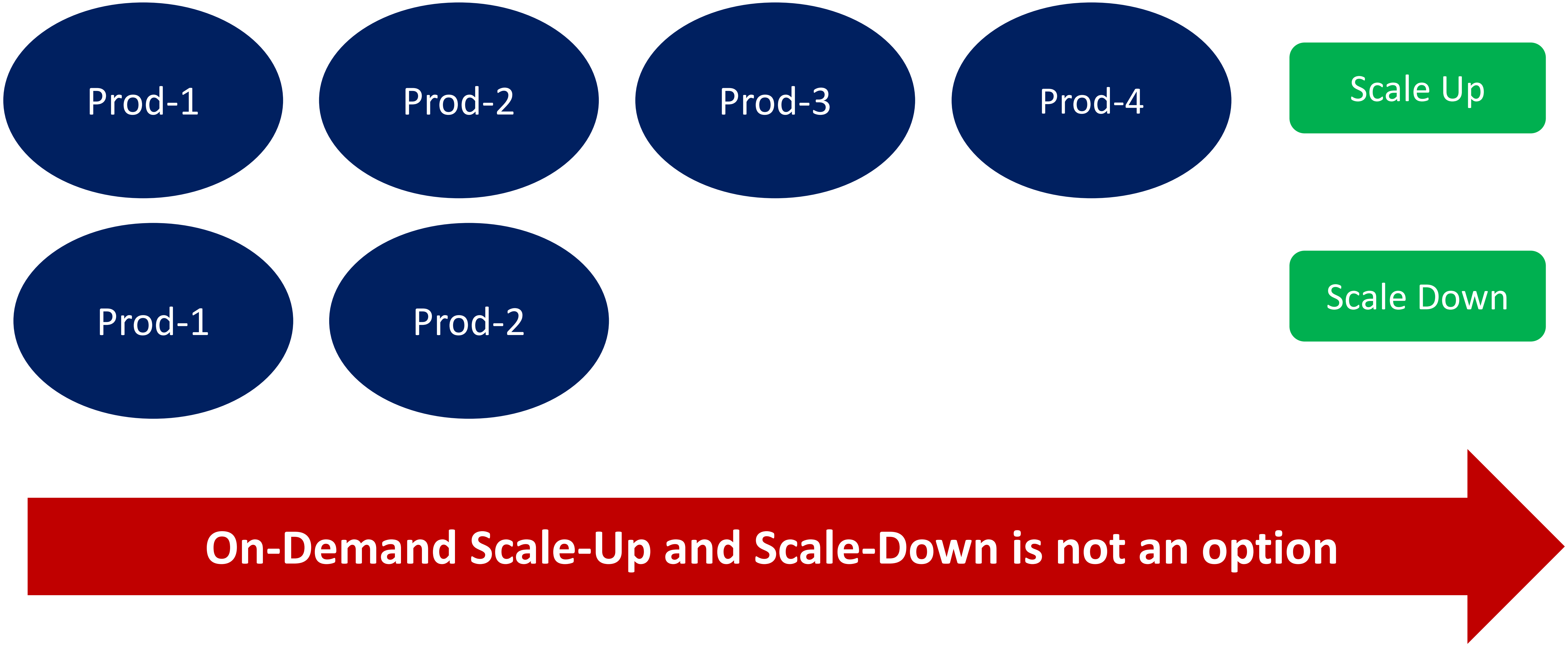
Many Problems at many places in manual process

# 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**

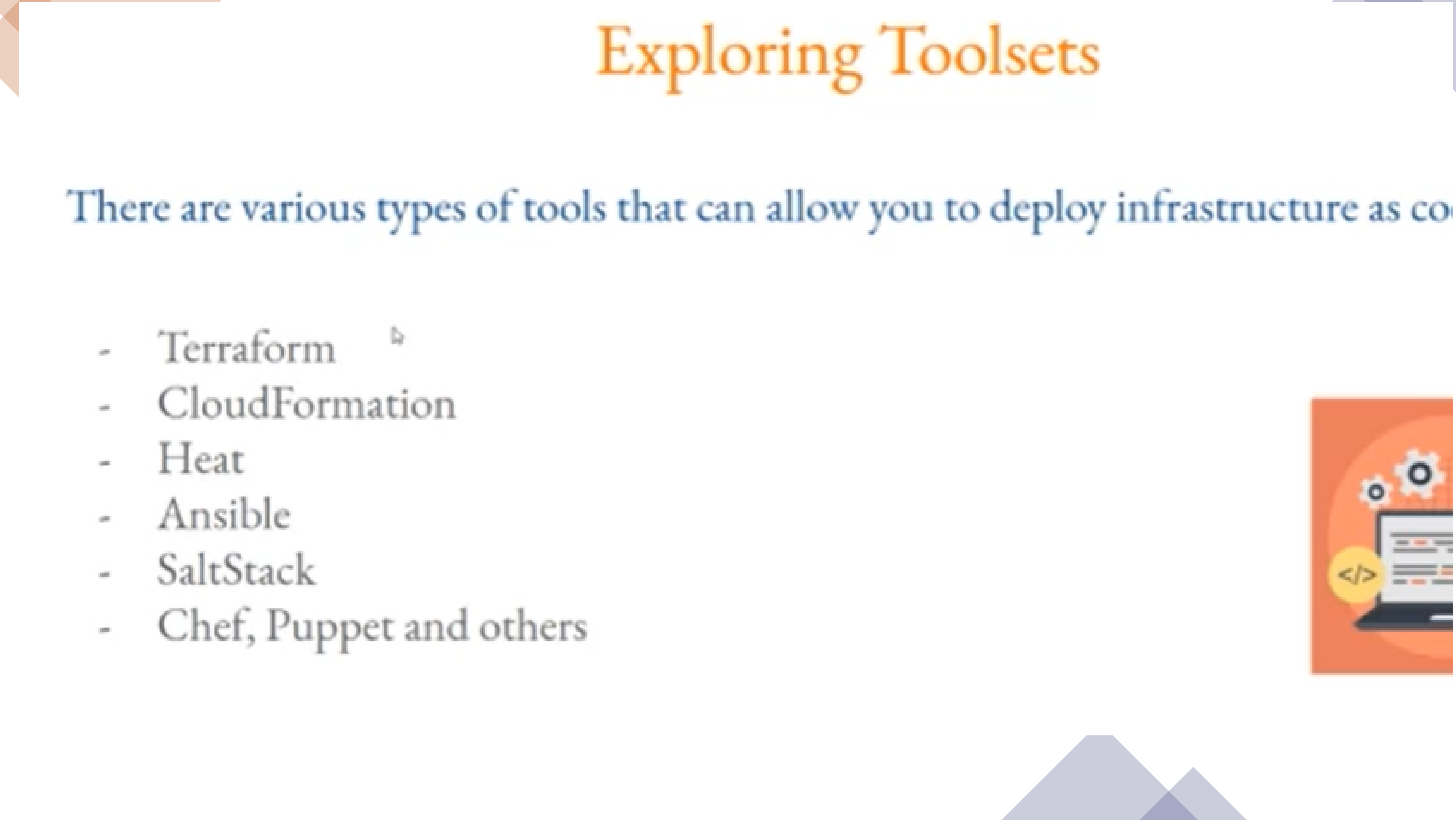
# What is Infrastructure as Code ?

# Configuration Management vs Infrastructure Orchestration

Ansible, Chef, Puppet are configuration management tools which means that they are primarily designed to install and manage software on existing servers.

Terraform, CloudFormation are the infrastructure orchestration tools which basically means they can provision the servers and infrastructure by themselves.

Configuration Management tools can do some degree of infrastructure provisioning, but the focus here is that some tools are going to be better fit for certain type of tasks.



# Exploring Toolsets

There are various types of tools that can allow you to deploy infrastructure as co

- Terraform
- CloudFormation
- Heat
- Ansible
- SaltStack
- Chef, Puppet and others

# Terraform

i) Supports multiple platforms, has hundreds of providers.

ii) Simple configuration language and faster learning curve.

iii) Easy integration with configuration management tools like Ansible.

iv) Easily extensible with the help of plugins.

v) Free !!!

# Supported Platforms

Terraform works on multiple platforms, these includes:

- Windows
- macOS
- Linux
- FreeBSD
- OpenBSD
- Solaris

## Advantages

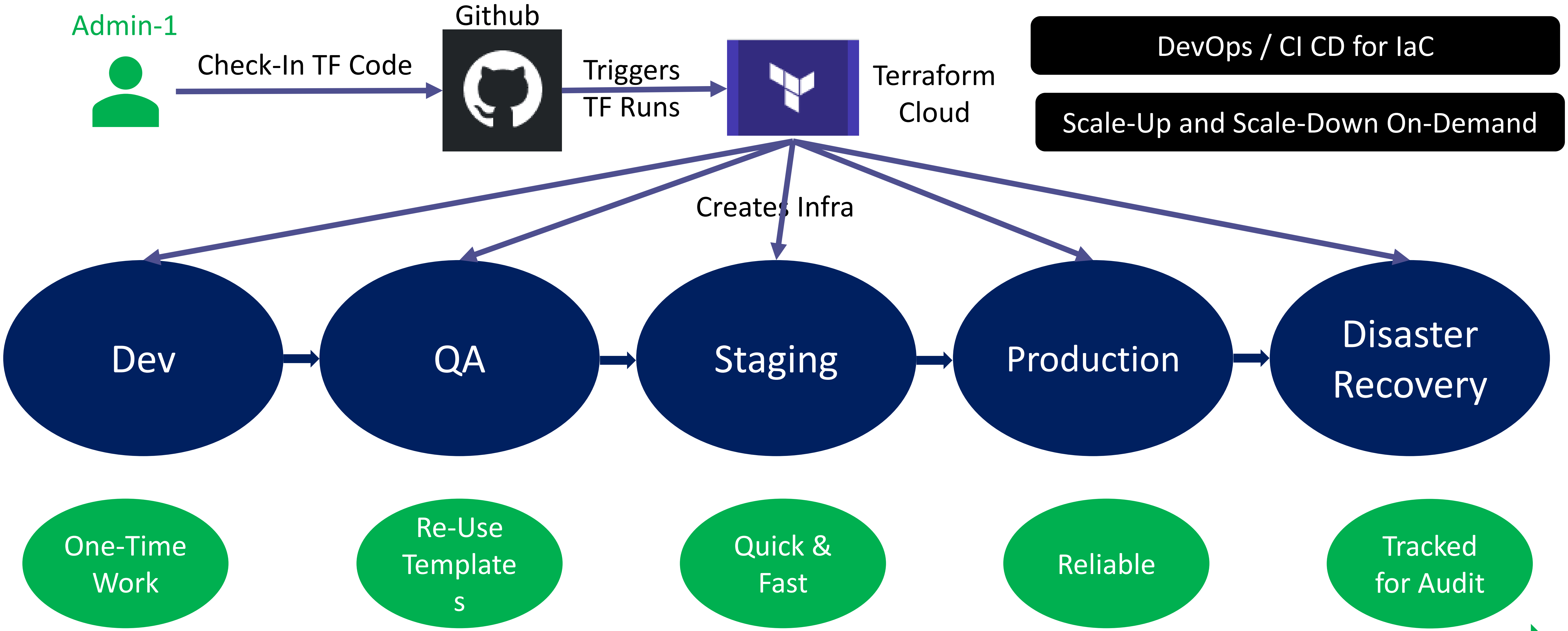
1.Terraform internally uses the DAG(direct acyclic graph) technique to get the best results.
2.Terraform supports a variety of cloud options, and switching providers is a breeze.
3.Because the whole infrastructure is managed as code, incremental resource changes are not a problem.
4.Supports scripts that span many regions. For instance, we can search for an ami in us-east-1 and use that information to build an ec2 instance in us-east-2.
5.Effective networking assistance. It might take months to build an on-premise data center, but using Terraform, it can be done in a matter of hours.
6.Integrates easily with the build and deployment processes.
7.Modular architecture.
8.State upkeep. Terraform will reconstruct any objects produced by it if another process removes them.
9.Allows for the import of existing resources to convert them to a Terraform state.

## Disadvantages

1.Currently under development. Each month, we release a beta version.
2.The concerns are more connected to Terraform’s (AWS) provider teams. For example, Terraform AWS’s quick sight does not yet support all features.
3.Technology with a narrow application. To write loops or if blocks, intuition is required. Nonetheless, several hacks are accessible online.
4.Specific configurations, such as the Terraform backend, are not accessible through var files. Therefore, either give the information in place or construct a backend-config block during Terraform’s initialization.
5.There is no error handling. This implies that we cannot utilize try-catch in the manner we do in other languages.
6.There is no way to roll back. As a result, we must delete everything and re-run if necessary.
7.A few things are prohibited from import.
8.Terraform does not support script generation from the state.
9.Terraform acknowledges that specific versions may include bugs.

5 Days

# Manage using IaC with Terraform



Admin-1

Check-In TF Code

Github

Triggers TF Runs

Terraform Cloud

DevOps / CI CD for IaC

Scale-Up and Scale-Down On-Demand

Creates Infra

Dev

QA

Staging

Production

Disaster Recovery

One-Time Work

Re-Use 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.

# Terraform
# Installation

# Terraform Installation

Terraform CLI

AWS CLI

VS Code Editor

Terraform plugin for VS Code

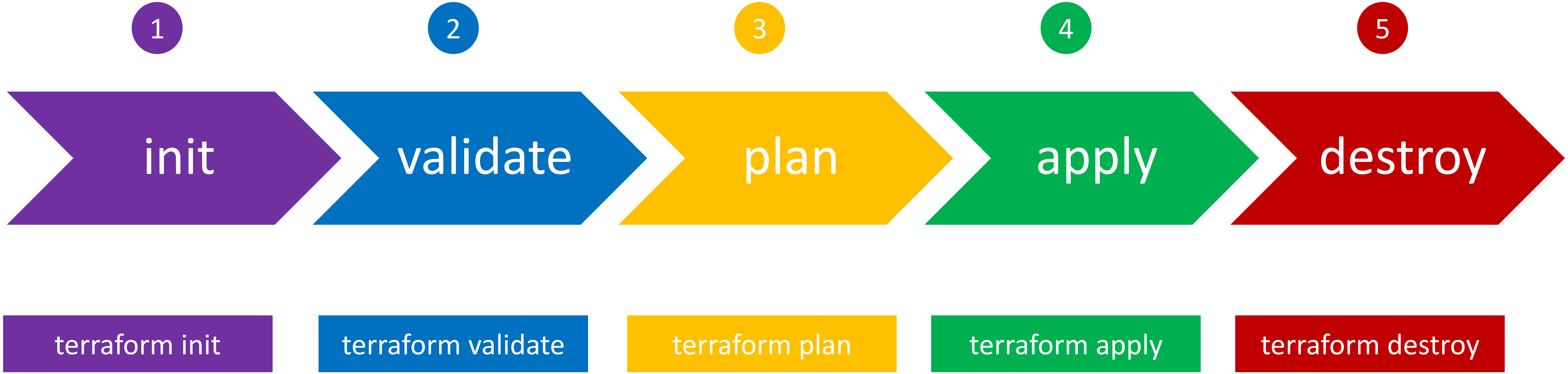
Mac OS

Windows OS

Linux OS

# Terraform
# Command Basics

# Terraform Workflow



1 init

2 validate

3 plan

4 apply

5 destroy

terraform init

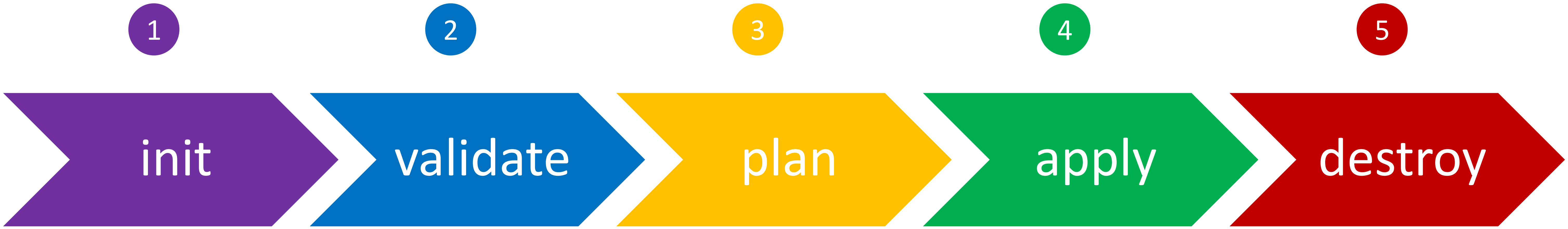
terraform validate

terraform plan

terraform apply

terraform destroy

# Terraform Workflow



## 1 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

## 2 validate

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

## 3 plan

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

## 4 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.

## 5 destroy

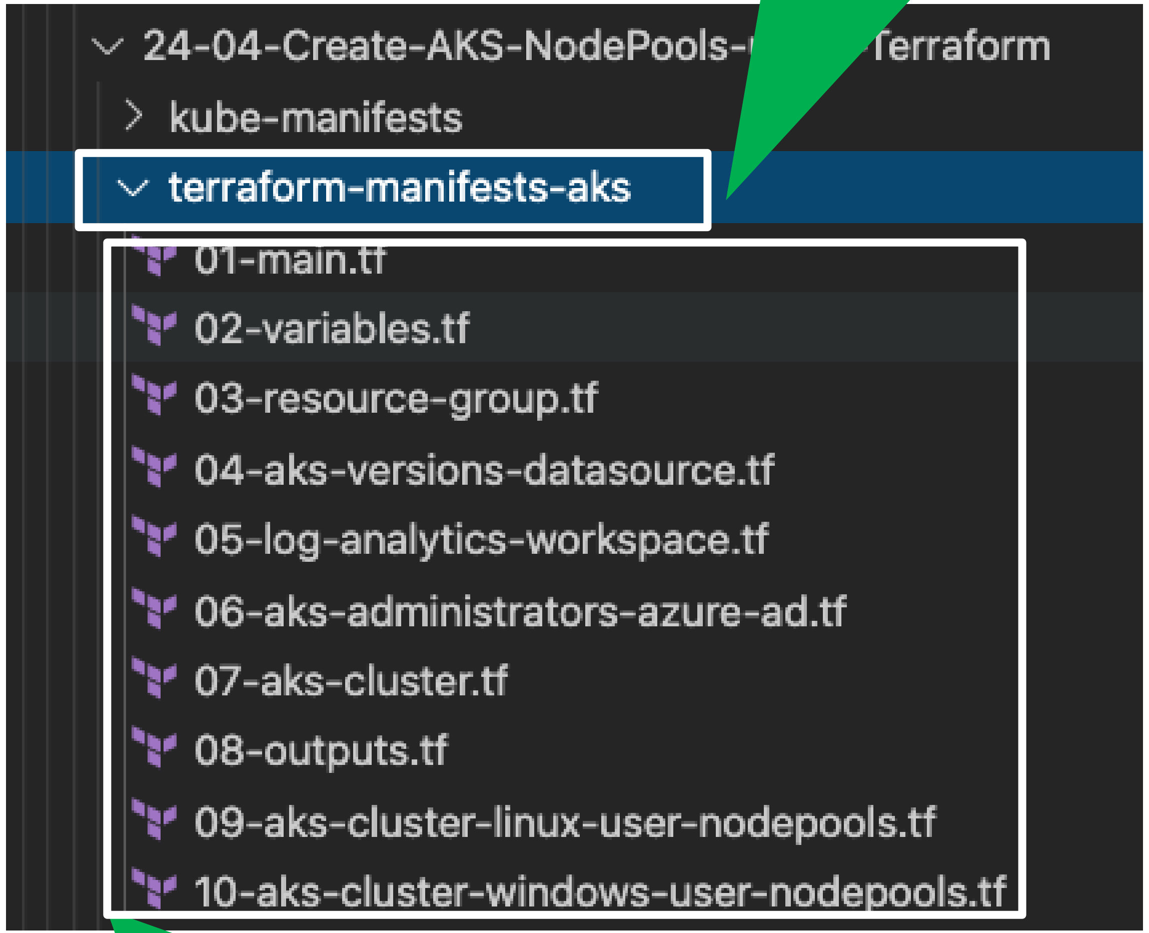
- Used to destroy the Terraform-managed infrastructure
- This will ask for confirmation before destroying.

# 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

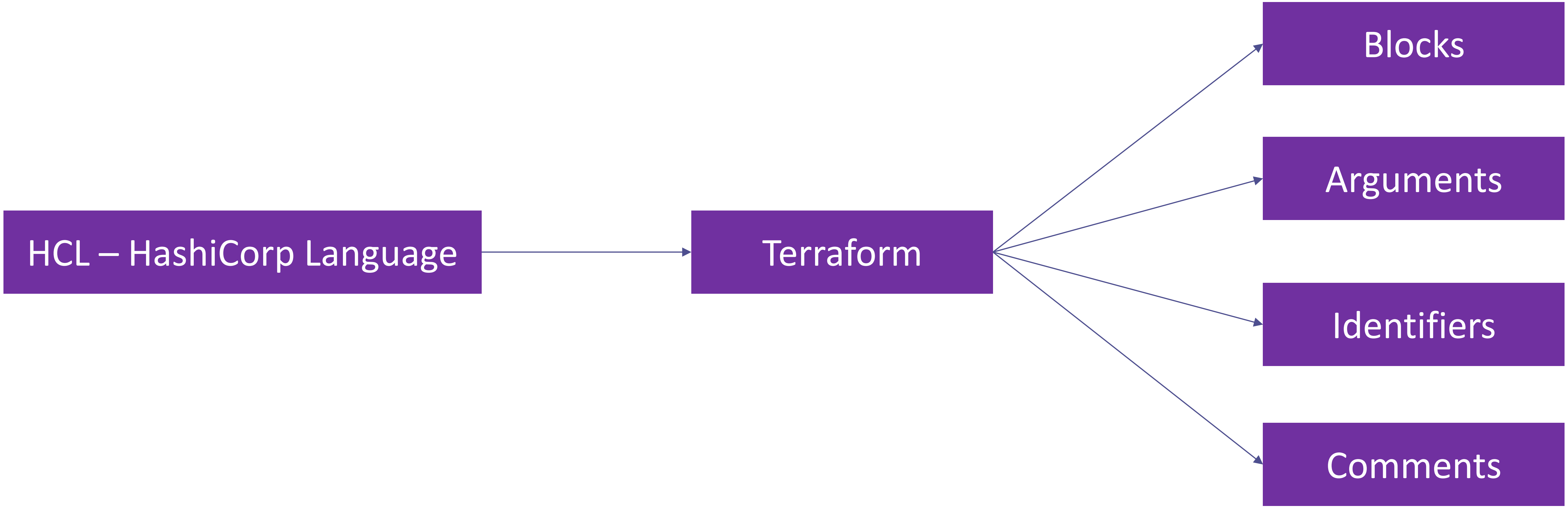
Terraform Working Directory



```
24-04-Create-AKS-NodePools-[illegible] Terraform
  kube-manifests
  terraform-manifests-aks
    01-main.tf
    02-variables.tf
    03-resource-group.tf
    04-aks-versions-datasource.tf
    05-log-analytics-workspace.tf
    06-aks-administrators-azure-ad.tf
    07-aks-cluster.tf
    08-outputs.tf
    09-aks-cluster-linux-user-nodepools.tf
    10-aks-cluster-windows-user-nodepools.tf
```

Terraform Configuration Files ending with .tf as extension

# Terraform Language Basics – Configuration Syntax



HCL – HashiCorp Language

Terraform

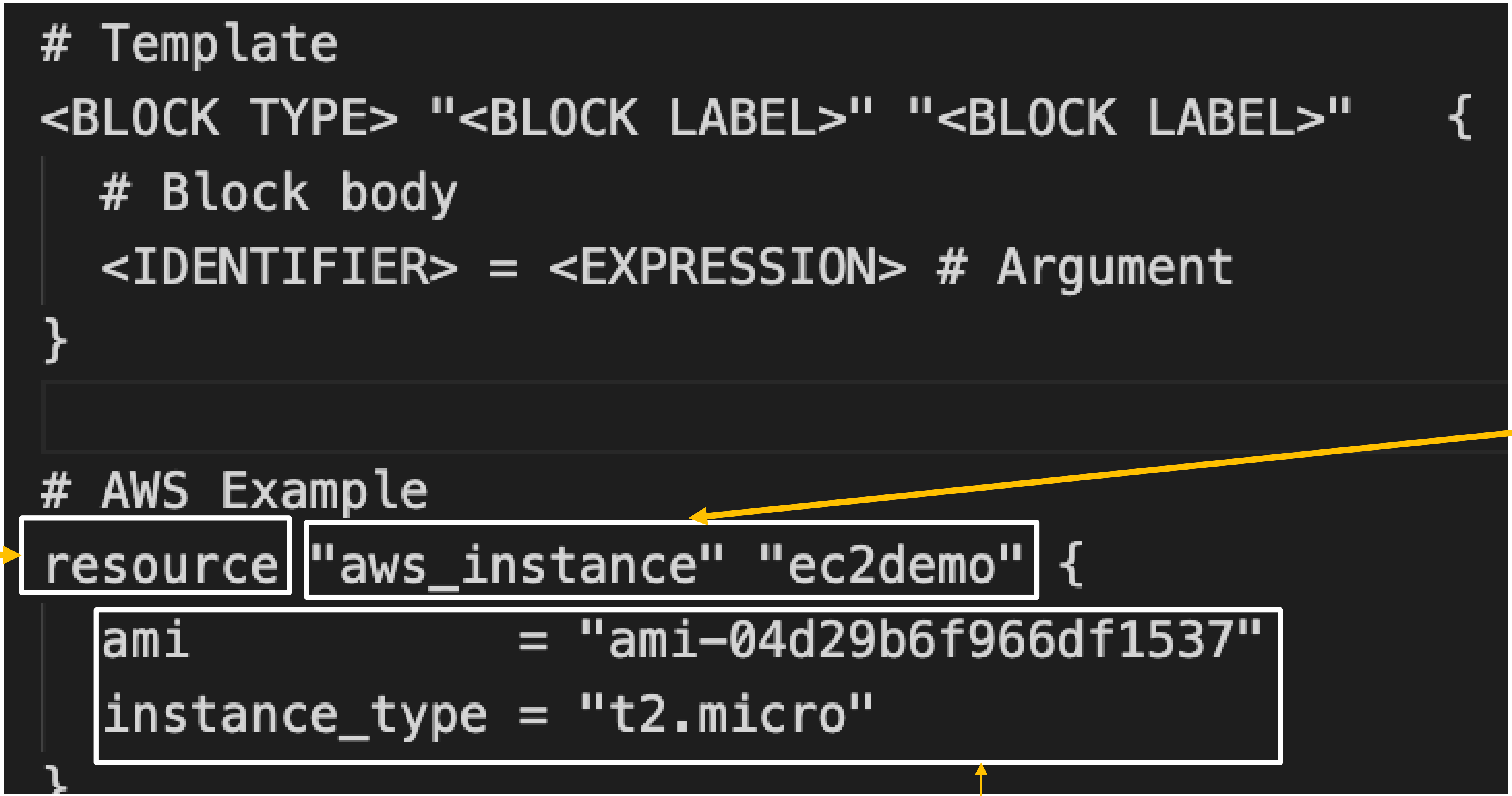
Blocks

Arguments

Identifiers

Comments

# 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

Arguments

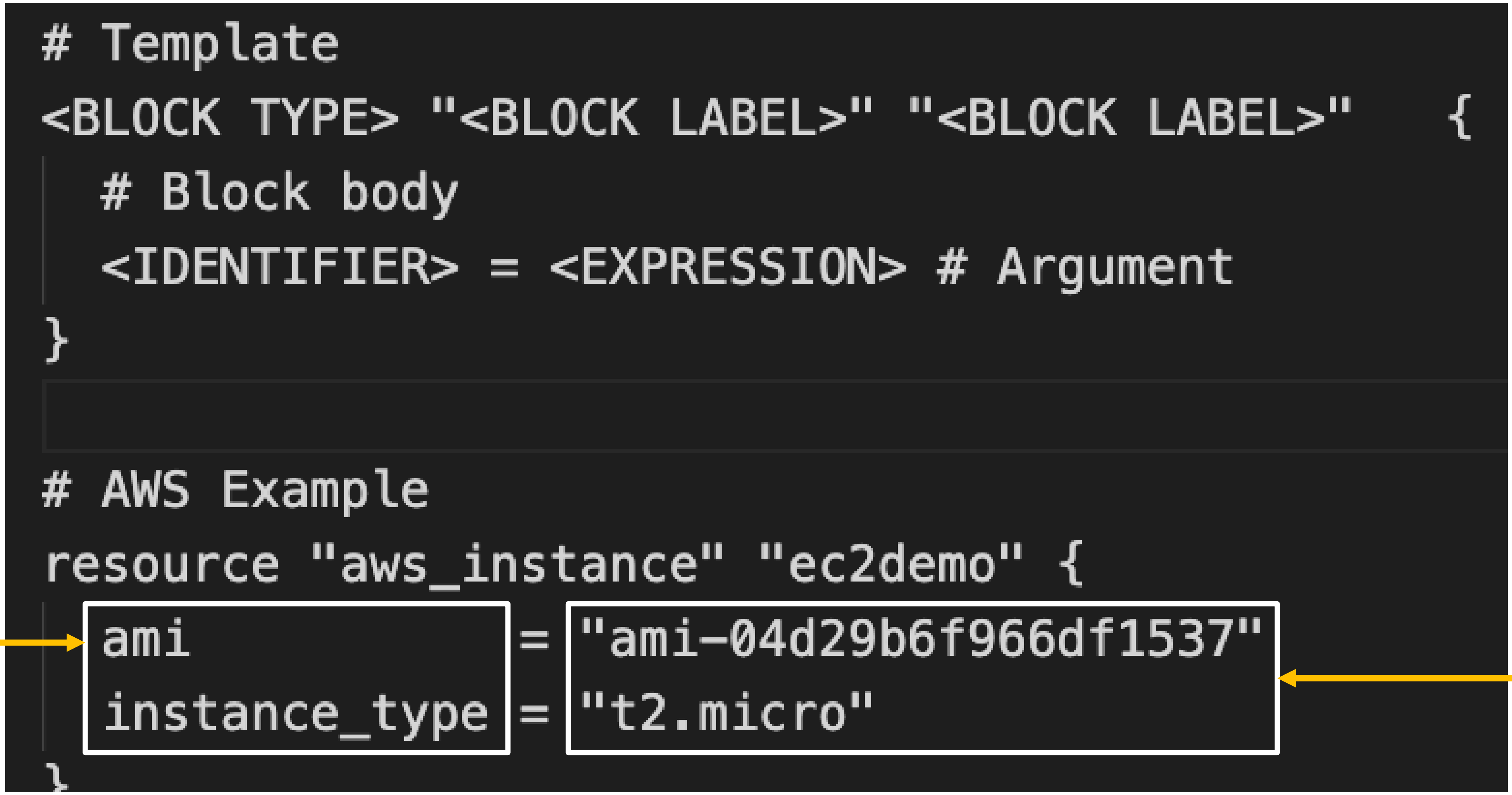
Top Level & Block inside Blocks

**Top Level Blocks:** resource, provider

**Block Inside Block:** provisioners, resource specific blocks like tags

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

# 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"
}
```

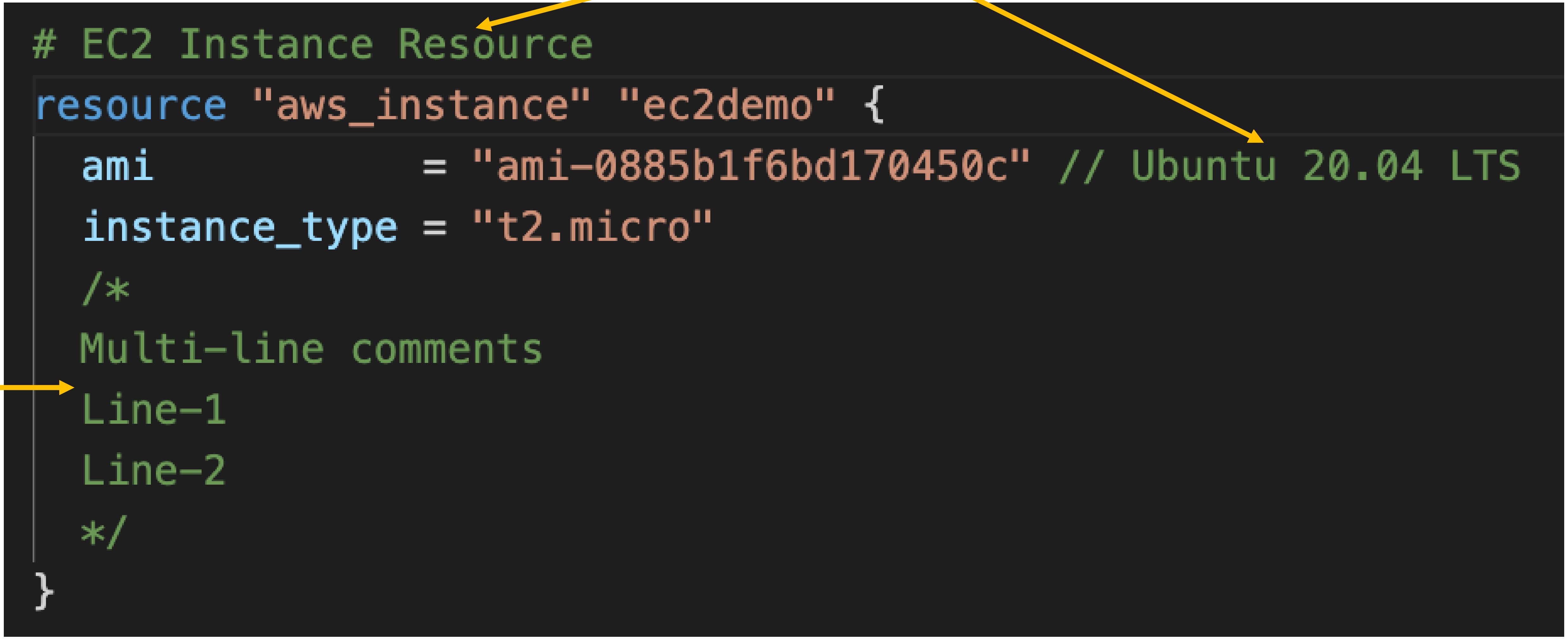
Argument
Name
[or]
Identifier

Argument
Value
[or]
Expression

# Terraform Language Basics – Configuration Syntax

Single Line Comments with # or //

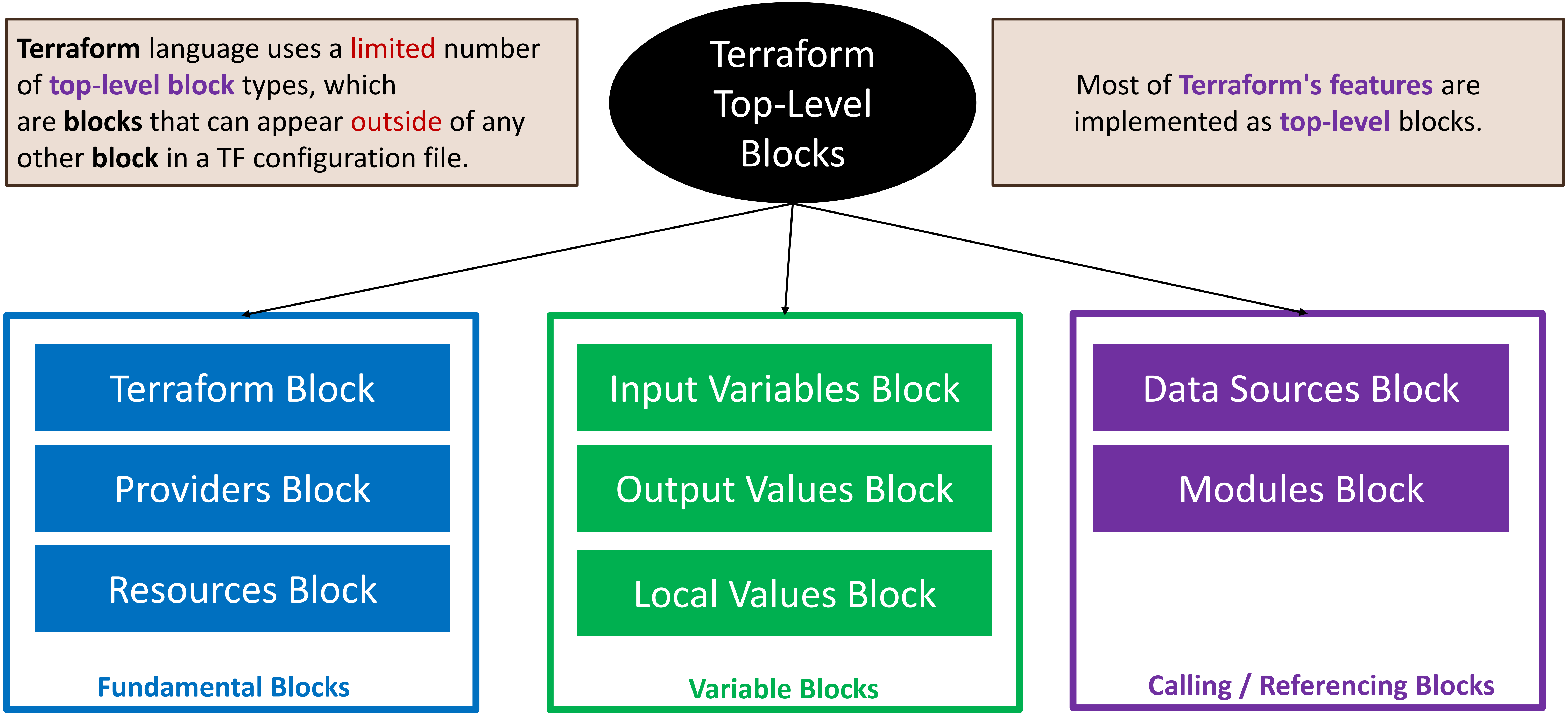
Multi-line comment



```
# 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
  */
}
```

**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.

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



Terraform Top-Level Blocks

**Fundamental Blocks**

- Terraform Block
- Providers Block
- Resources Block

**Variable Blocks**

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

**Calling / Referencing Blocks**

- Data Sources Block
- Modules Block

# Terraform
# Fundamental Blocks

# Terraform Basic Blocks

## Terraform Block

- Special block used to configure some behaviors
- Required Terraform Version
- List Required Providers
- Terraform Backend

## 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

c1-versions.tf

## 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

c2-resource-name.tf

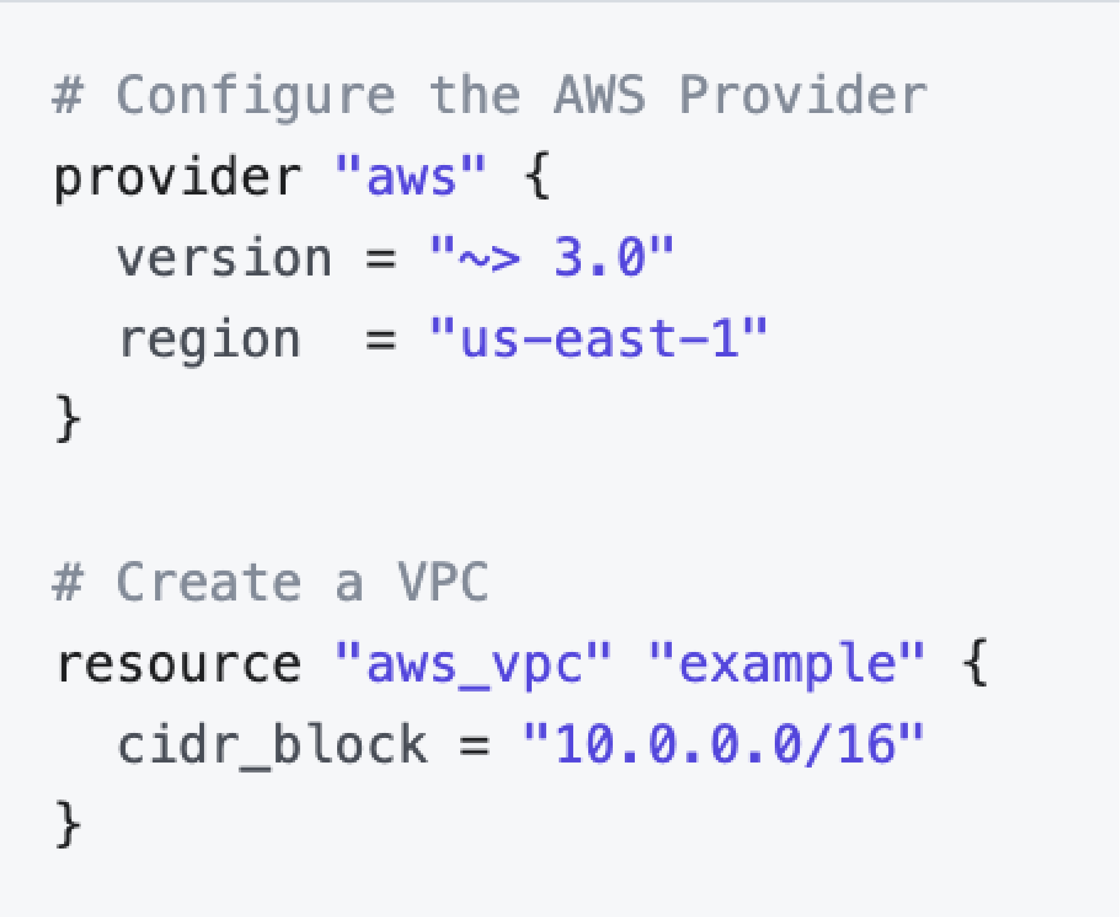
# Terraform Block

# 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

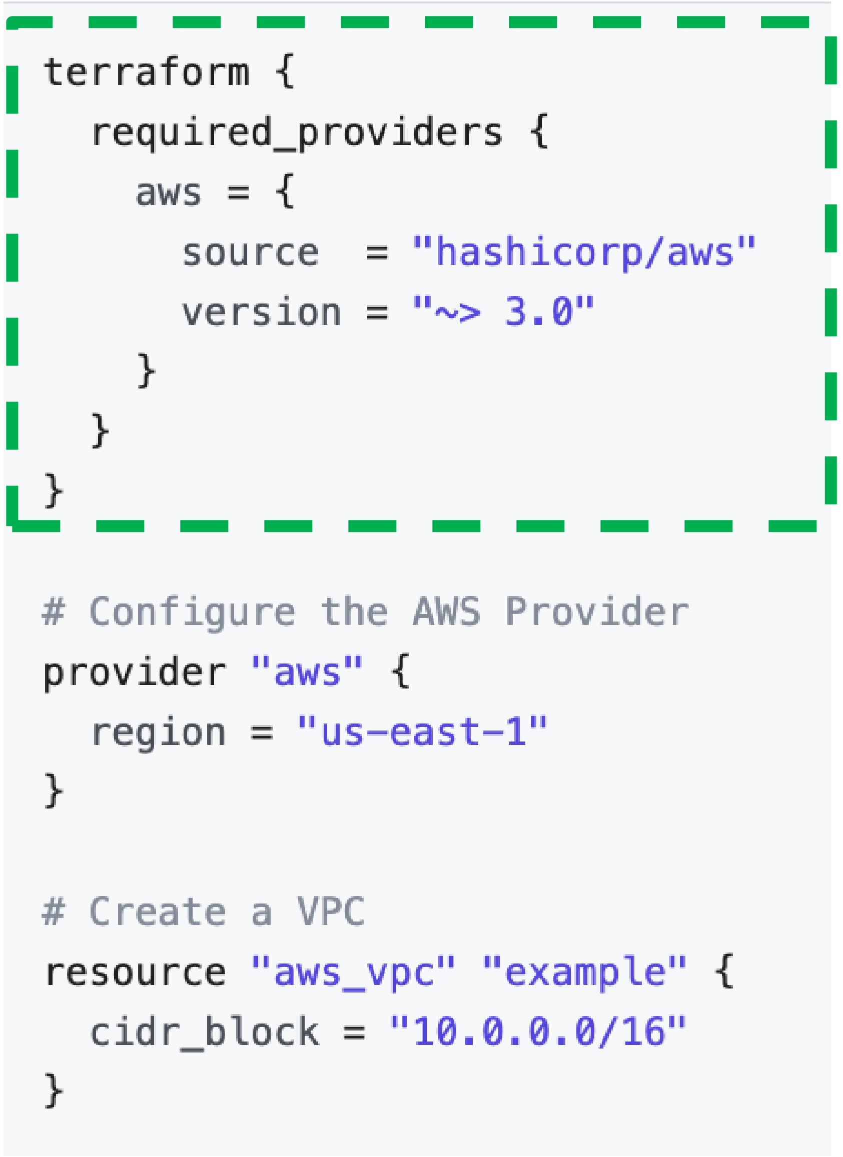
Terraform 0.12 and earlier:



```
# 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:

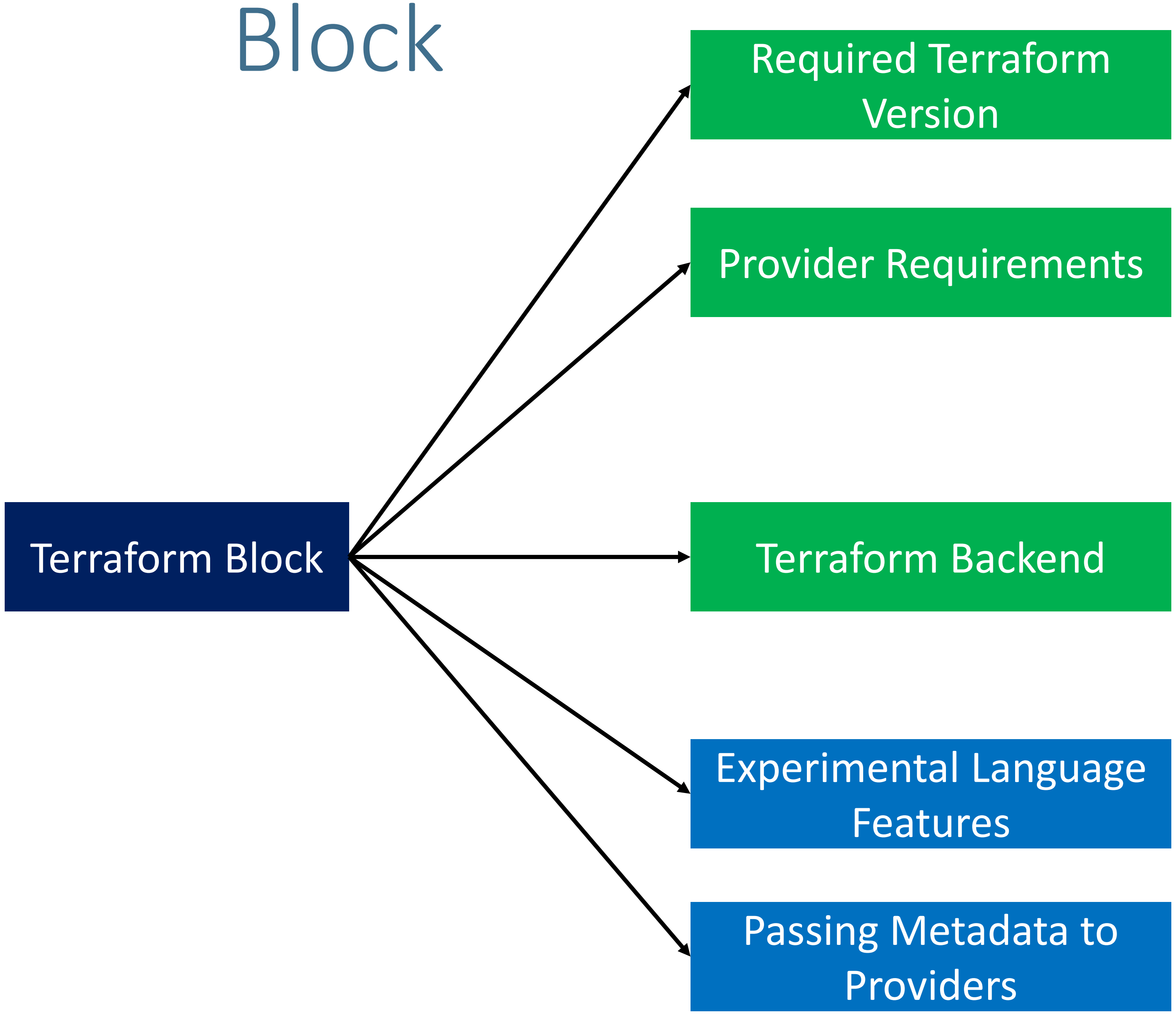


```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

# 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



Terraform Block

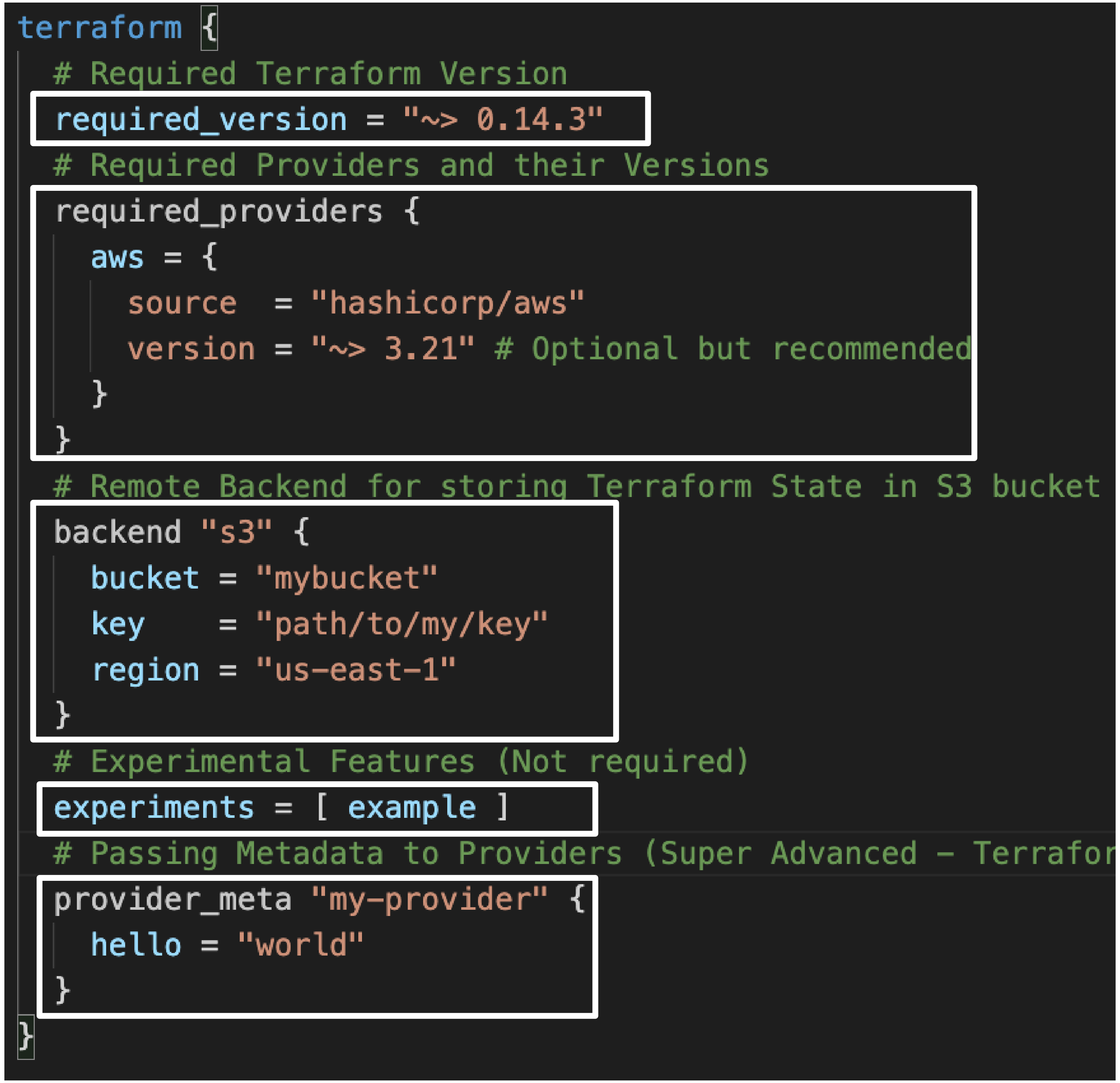
Required Terraform Version

Provider Requirements

Terraform Backend

Experimental Language Features

Passing Metadata to Providers



```
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"
  }
}
```