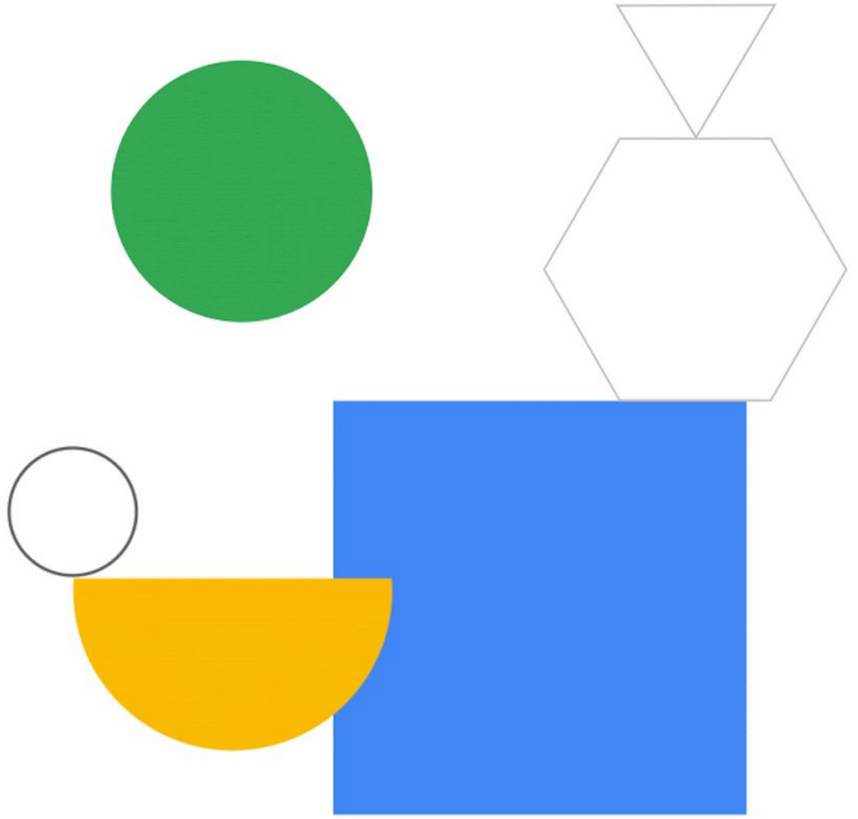
Google Cloud

Terms and Concepts



Objectives

Upon completion of this module, you will be able to:

01 Explain the Terraform workflow.

02 Create basic configuration files within Terraform.

03 Explain the purpose of a few Terraform commands.

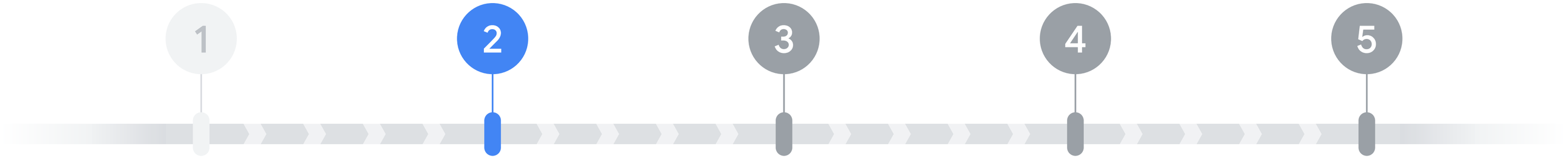
04 Describe the Terraform Validator tool.

05 Create, update, and destroy Google Cloud resources using Terraform.

Topics

01	The Author phase
02	Terraform commands
03	Terraform Validator Tool

Terraform workflow



Scope	Author	Initialize	Plan	Apply
Confirm the resources required for a project.	Author the configuration files based on the scope.	Download the provider plugins, and initialize the module.	View execution plan for resources created, modified, or destroyed.	Create actual infrastructure resources.

Terraform Workflow

Terraform directory

- Terraform uses configuration files to declare an infrastructure element.
- The configuration is written in terraform language with a .tf extension.
- A configuration consists of:
  - A root module/ root configuration
  - Zero or more child modules
  - Variables.tf (optional but recommended)
  - Outputs.tf (optional but recommended)
  - Terraform.tfvars (optional but recommended)
- Terraform commands are run on the working directory.

```
-- main.tf
   -- servers/
      -- main.tf
      -- providers.tf
      -- variables.tf
      -- outputs.tf
      -- terraform.tfvars
```

Root module

Child module

HashiCorp Configuration Language (HCL)

- Terraform's configuration language for creating and managing API-based resources
- Configuration language, not a programming language.
- Includes limited set of primitives such as variables, resources, outputs and modules.
- Does not include traditional statements or control loops.

Syntax

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

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- Comments

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

```
resource "google_compute_network" "default" {
   # A custom mode VPC network resource
   name                    = "mynetwork"
   auto_create_subnetworks  = false
}
```

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- Comments

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

```
resource "google_compute_network" "default" {
   # custom mode network definition
   name                    = "mynetwork"
   auto_create_subnetworks  = false
}
```

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- Comments

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

```
resource "google_compute_network" "default" {
   # custom mode network definition
   name                    = "mynetwork"
   auto_create_subnetworks  = false
}
```

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- Comments

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

```
resource "google_compute_network" "default" {
   # custom mode network definition
  name                            = "mynetwork"
  auto_create_subnetworks   = false
}
```

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- Comments

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

```
resource "google_compute_network" "default" {
   # custom mode network definition
   name                    = "mynetwork"
   auto_create_subnetworks  = false
}
```

HCL syntax

- Blocks
- Arguments
- Identifiers
- Expressions
- **Comments**

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

```
resource "google_compute_network" "default" {
  # custom mode network definition
  name                    = "mynetwork"
  auto_create_subnetworks  = false
}
```

Resources

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- Resources are code blocks that define the infrastructure components.

  Example: Cloud Storage Bucket

- Terraform uses the **`resource type`** and the **`resource name`** to identify an infrastructure element.

```
resource "resource_type" "resource_name" {
  #Resource specific arguments
}
```

Example for resources

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- The keyword **resource** is used to identify the block as the cloud infrastructure component.
- Resource type is **google_storage_bucket**.
- The resource name is **example-bucket**.

```
resource "google_storage_bucket" "example-bucket" {
  name          = "<unique-bucket-name>"
  location      = "US"
}
```

Resource arguments

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- The arguments differ based on the resource type.
- Some arguments are required, others are optional.

```
resource "google_storage_bucket" "example-bucket" {
   name          = "<unique-bucket-name>" //Required
   location      = "US"
 }

resource "google_compute_instance" "my_instance" {
  name         = "test"
  machine_type = "e2-medium"
  zone         = "us-central1-a"
  boot_disk {
      initialize_params {
        image = "debian-cloud/debian-9"
        }
  }
  network_interface {
    network = "default"
    }
  }
}
```

Provider (1 of 2)

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- Terraform downloads the provider plugin in the root configuration when the provider is declared.
- Providers expose specific APIs as Terraform resources and manage their interactions.

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
      version = "4.23.0"
    }
  }
}

provider "google" {
  # Configuration options
   project = <project_id>
   region  = "us-central1"
}
```

Provider (2 of 2)

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- Provider configurations belong in the root module of a Terraform configuration.
- Arguments such as project and region can be declared within the provider block.

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "4.23.0"
    }
  }
}

provider "google" {
  # Configuration options
  project = <project_id>
  region  = "us-central1"
}
```

Provider versions

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- The version argument is optional, but recommended.
- The version argument is used to constrain the provider to a specific version or a range of versions.

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = "4.23.0"
    }
  }
}

provider "google" {
  # Configuration options
}
```

Variables

- Parameterize resource arguments to eliminate hard coding its values

  Example: Region, project ID, zone, etc.
- Define a resource attribute at run time or centrally in a file with a .tfvars extension.

Storage bucket

main.tf

Declare storage bucket

```
location
unique_name
storage_class
versioning
```

variables.tf

Parameterize resource attribute

```
variable location {
..
}
```

terraform.tfvars

Define values at run time or in a file

```
location= "US"
storage_class="STANDARD"
```

CLI

```
$terraform apply

location= "US"
storage_class=”STANDARD”
```

Outputs values

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
```

- Output values are stored in `outputs.tf` file.
- Output values expose values of resource attributes.

```
output "bucket_URL" {
  value = google_storage_bucket.mybucket.URL
}
```

```
#terraform apply
Google_storage_bucket.mybucket: Creating...
Google_storage_bucket.mybucket: Creating complete after 1s []

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

bucket_URL = “https://storage.googleapis.com/my-gallery/..
```

State

```
-- main.tf
-- providers.tf
-- variables.tf
-- outputs.tf
-- terraform.tfvars
-- terraform.tfstate
```

- Terraform saves the state of resources it manages in a state file.
- The state file can be stored:
    - Locally (default)
    - Remotely in a shared location

You do not modify this file.

```
{
  "version": 4,
  "terraform_version": "1.0.11",
  "serial": 3,
  "lineage": "822c3d96-0500-29cd-68e3-13101f2846f0",
  "outputs": {
    "vm_name": {
      "value": "terraform-test",
      "type": "string"
    }
  },
  "resources": [
    {
      "mode": "managed",
      "type": "google_compute_instance",
      "name": "default",
      "provider":
"provider[\"registry.terraform.io/hashicorp/google\"]",
      "instances": [
        {
```

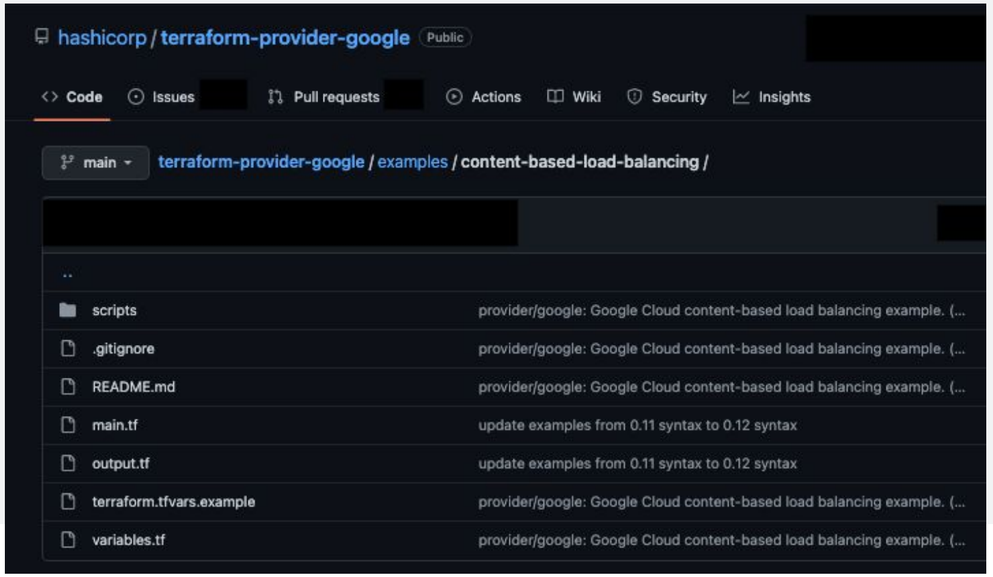
Modules

A Terraform module is a set of Terraform configuration files in a single directory.

- It is the primary method for code reuse in Terraform.
- There are 2 kinds of sources:
  - Local: Source within your directory
  - Remote: Source outside your directory.

```
-- instances/
   -- main.tf
   -- variables.tf
   -- outputs.tf
   -- terraform.tfvars
```

modules



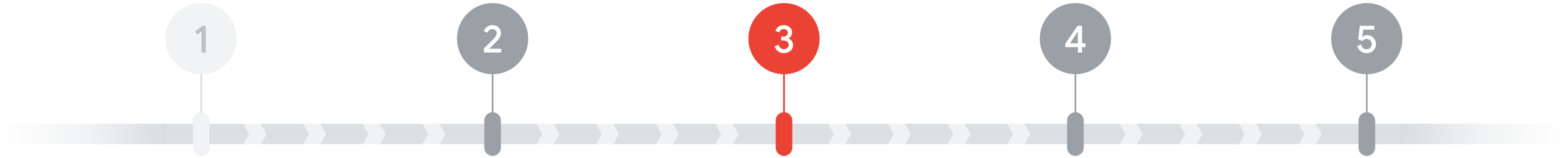
Topics

01	The Author phase
02	Terraform commands
03	Terraform Validator tool

Terraform commands

`terraform init`	Initialize the provider with plugin
`terraform plan`	Preview of resources that will be created after terraform apply
`terraform apply`	Create real infrastructure resources
`terraform destroy`	Destroy infrastructure resources
`terraform fmt`	Auto format to match canonical conventions

Initialize Terraform using `terraform init`



Scope	Author	Initialize	Plan	Apply
Confirm the resources required for a project.	Author the configuration files based on the scope.	Download the provider plugins, and initialize the module.	View execution plan for resources created, modified, or destroyed.	Create actual infrastructure resources.

Terraform Workflow

Initialize Phase:

`terraform init` downloads the provider plugins

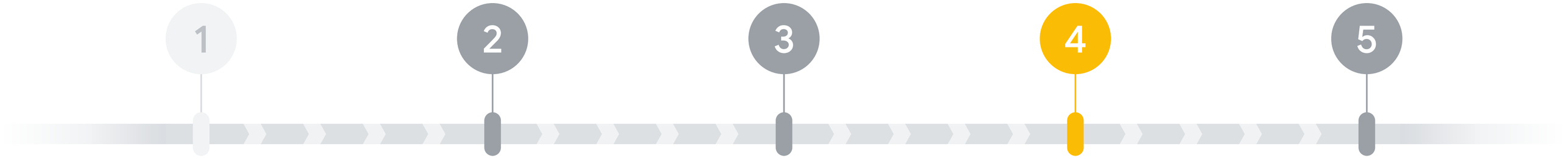
```
-- main.tf
    -- servers/
        -- main.tf
        -- variables.tf
        -- outputs.tf
        -- terraform.tfvars
```

```
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
    }
  }
}
```

```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/google...
- Installing hashicorp/google v4.21.0...
- Installed hashicorp/google v4.21.0 (signed by HashiCorp)
..
Terraform has been successfully initialized!
```

Preview resource action using `terraform plan`



Scope	Author	Initialize	Plan	Apply
Confirm the resources required for a project.	Author the configuration files based on the scope.	Download the provider plugins, and initialize the module.	View execution plan for resources created, modified, or destroyed.	Create actual infrastructure resources.

Terraform Workflow

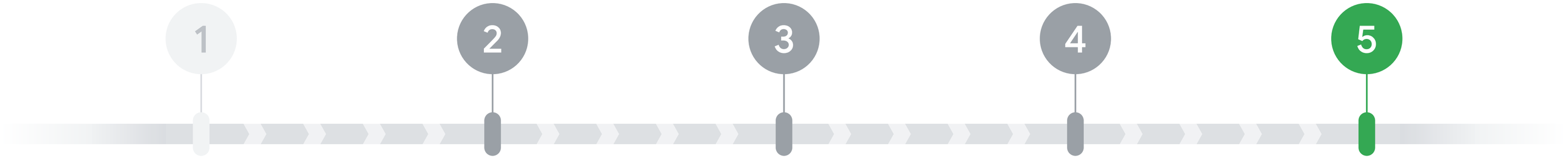
Plan Phase:

`terraform plan` creates an execution plan

```
resource "google_storage_bucket" "example-bucket"{
  name          = "student0313ab04569a94"
  location      = "US"
}
```

```
$terraform plan
Terraform will perform the following actions:
  # google_storage_bucket.example-bucket will be created
  + resource "google_storage_bucket" "example-bucket" {
      + force_destroy               = false
      + id                          = (known after apply)
      + location                    = "US"
      + name                        = "student0313ab04569a94"
      + storage_class               = "STANDARD"
      + uniform_bucket_level_access = (known after apply)
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

Executes the actions proposed in a Terraform plan using `terraform apply`



Scope	Author	Initialize	Plan	Apply
Confirm the resources required for a project.	Author the configuration files based on the scope.	Download the provider plugins, and initialize the module.	View execution plan for resources created, modified, or destroyed.	Create actual infrastructure resources.

Terraform Workflow

Apply Phase

`terraform apply` executes the plan

```
resource "google_storage_bucket" "example-bucket"{
  name     = "student0313ab04569a94"
  location = "US"
}
```

```
$terraform apply
Terraform will perform the following actions:
  # google_storage_bucket.example-bucket will be created
  + resource "google_storage_bucket" "example-bucket" {
      + force_destroy               = false
      + id                          = (known after apply)
      + location                    = "US"
      + name                        = "student0313ab04569a94"
      …
Apply changes: yes
google_storage_bucket.example-bucket: Creating...
google_storage_bucket.example-bucket: Creation complete after 1s [id=student0313ab04569a94]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

Code conventions

Formatting best practices:

- Separate meta arguments from the other arguments.
- Use two spaces for indentation.
- Align values at the equal sign.
- Place nested blocks below arguments.
- Separate blocks by one blank line.

[Style conventions]

```
resource “google_compute_instance” “my-instance” {
boot_disk { #nested arguments above
       initialize_params {
         image = "debian-cloud/debian-9"
       }
     }
   count = 2 #meta-argument in between
   name = "test"
   machine_type="e2-micro" #unaligned equal signs
   ..
}
```

```
resource “google_compute_instance” “my-instance” {
   count        = 2 #meta-argument first

   name         = "test"
   machine_type = "e2-micro" #align equal signs
   boot_disk { #nested arguments below
       initialize_params {
         image  = "debian-cloud/debian-9"
       }
     }..
}
```

terraform fmt

Before `terraform fmt`:

```
resource “google_compute_instance” “my-instance” {
   boot_disk { #nested arguments above
       initialize_params {
         image="debian-cloud/debian-9"
       }
   }
   count = 2 #meta-argument in between
   name = "test"
   machine_type="e2-micro" #unaligned equal signs
   ..
```

After `terraform fmt`:

```
resource “google_compute_instance” “my-instance” {
   count            = 2 #meta-argument first
   name             = "test"
   machine_type     = "e2-micro" #align equal signs
   #line space before a nested block

   boot_disk { #nested arguments below
       initialize_params {
         image = "debian-cloud/debian-9"
       }
     }..
```

terraform destroy command

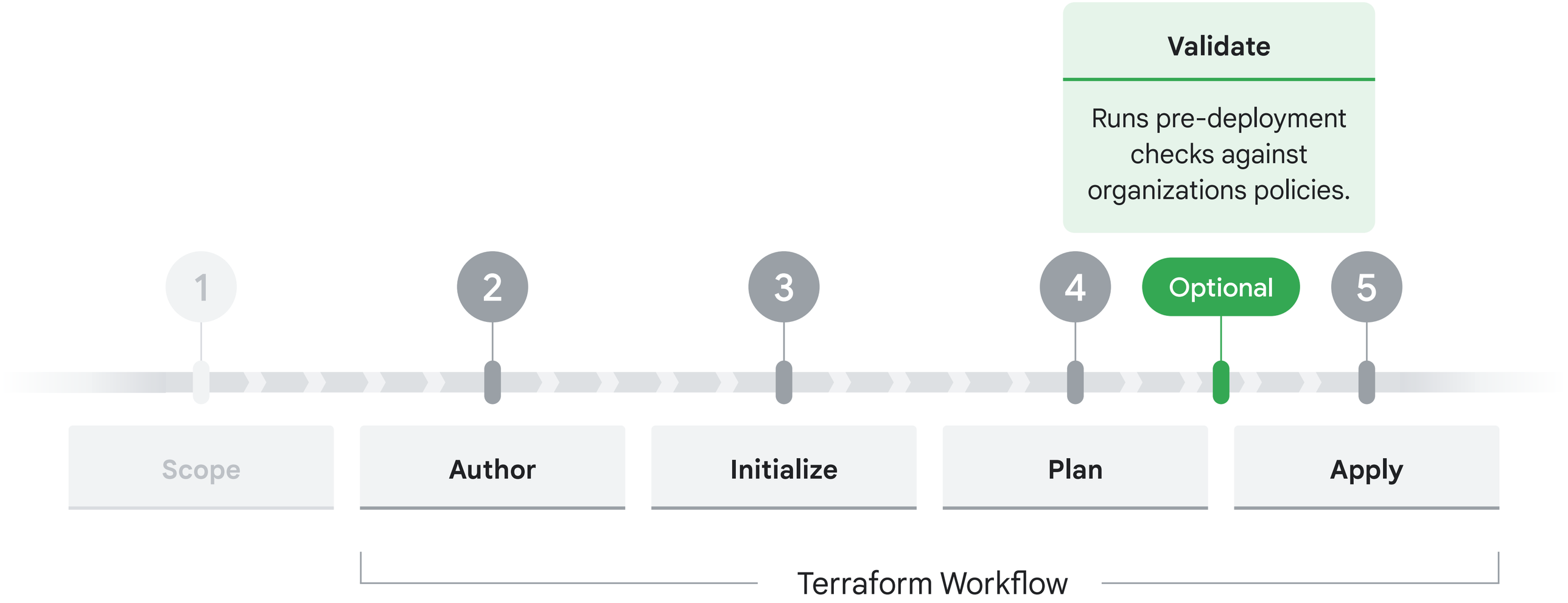
```
$ terraform destroy
google_storage_bucket.example-bucket: Refreshing state... [id=student0313ab04569a94]
..
Terraform will perform the following actions:
  # google_storage_bucket.example-bucket will be destroyed
  - resource "google_storage_bucket" "example-bucket" {
      - force_destroy                 = false -> null
      - id                            = "student0313ab04569a94" -> null
      - location                      = "US" -> null
      - name                          = "student0313ab04569a94" -> null
      ..
    }
Plan: 0 to add, 0 to change, 1 to destroy.
..
google_storage_bucket.example-bucket: Destroying... [id=student0313ab04569a94]
google_storage_bucket.example-bucket: Destruction complete after 0s
…
Destroy complete! Resources: 1 destroyed.
```

The **terraform destroy** command will destroy all the resources and its associated data from the main directory.

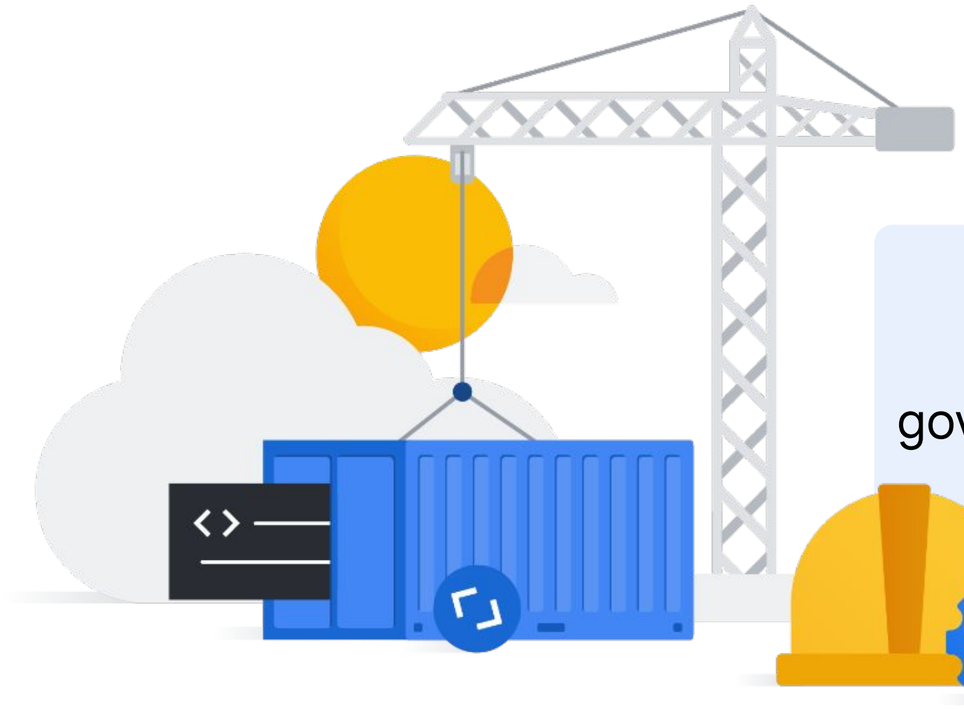
Topics

01	The Author phase
02	Terraform commands
03	Terraform Validator Tool

The validate phase



Why use the Terraform validator tool?



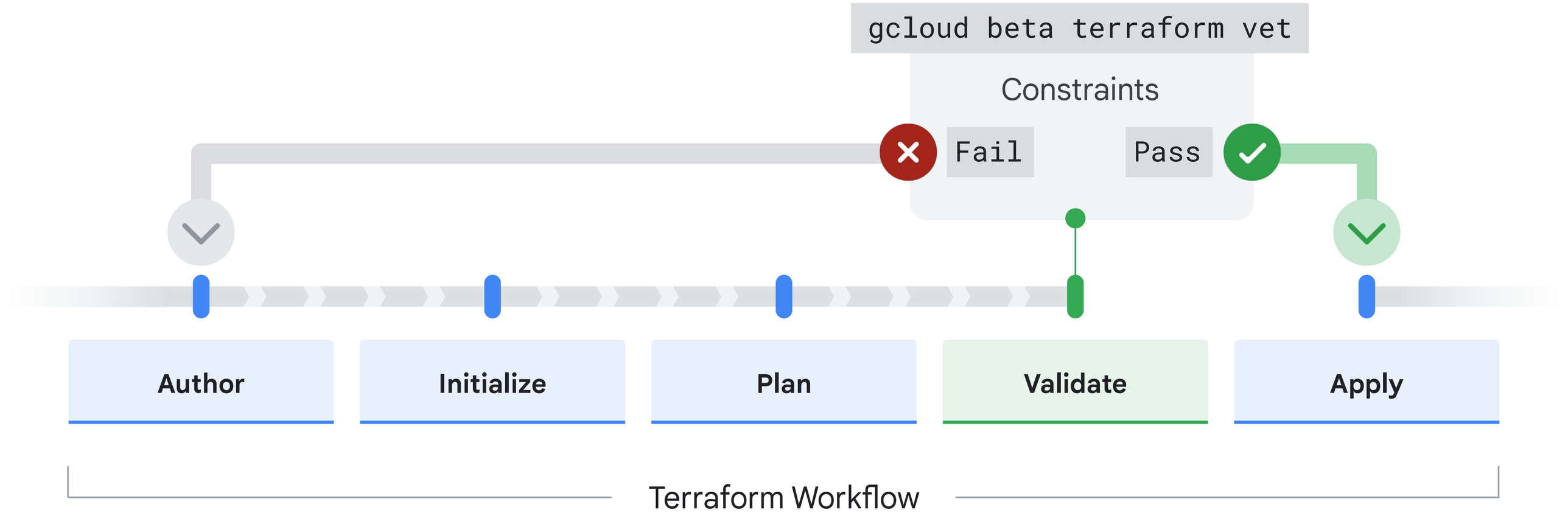
Security and governance team set constraints.

Automate validation

Governance policies

Security violation

gcloud beta terraform vet



Terraform Validator uses

01

Platform teams can add guardrails to infrastructure CI/CD pipelines to ensure all changes are validated.

02

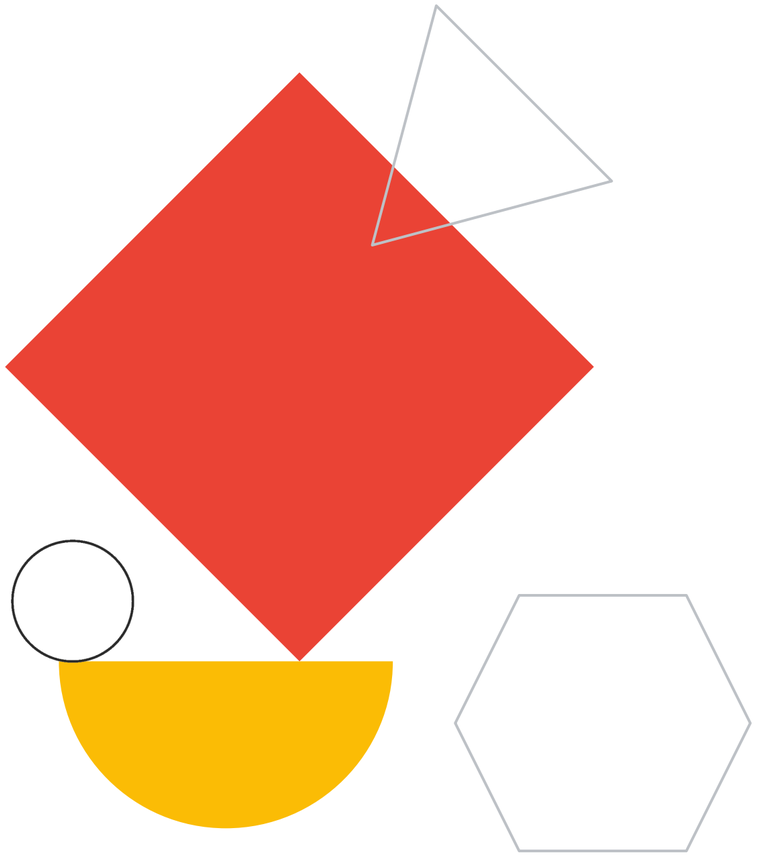
Application teams and developers can validate Terraform configuration with organization's central policy library.

03

Security teams can create a centralized policy library to identify and prevent policy violations.

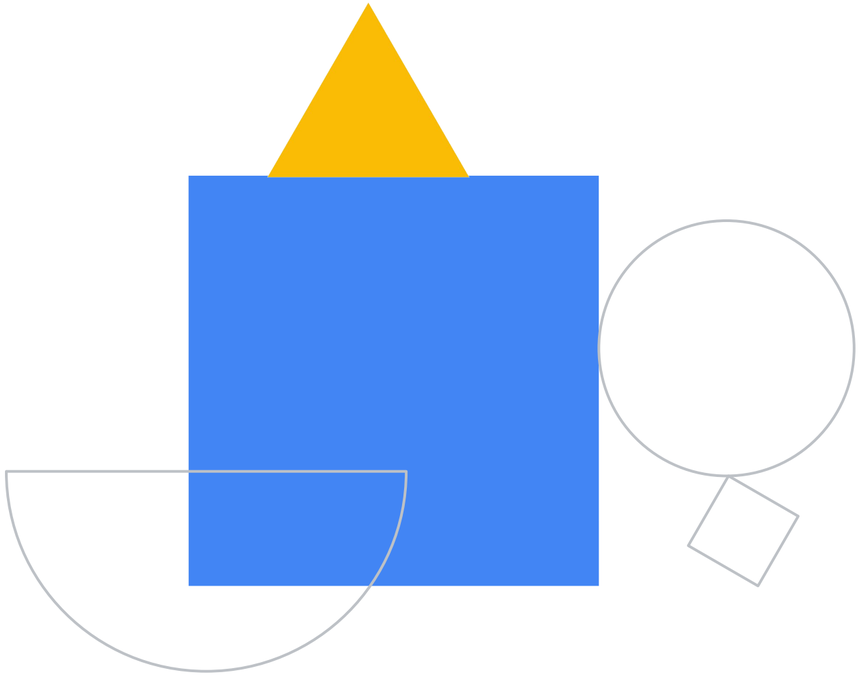
Demo

Demonstrate Terraform Workflow



Lab

Infrastructure as Code with Terraform



Quiz

Quiz | Question 1

Question

In which phase of the Terraform workflow can you run pre-deployment checks against the policy library?

A. Plan

B. Scope

C. Validate

D Initialize

Quiz | Question 1

Answer

In which phase of the Terraform workflow can you run pre-deployment checks against the policy library?

A. Plan

B. Scope

C. Validate

D Initialize

Quiz | Question 2

Question

Which command creates infrastructure resources?

A. terraform apply

B. terraform plan

C. terraform fmt

D terraform init

Quiz | Question 2

Answer

Which command creates infrastructure resources?

A. terraform apply

B. terraform plan

C. terraform fmt

D terraform init

Quiz | Question 3

Question

In which phase of the Terraform workflow do you write configuration files based on the scope defined by your organization?

A. Author

B. Initialize

C. Plan

D Scope

Quiz | Question 3

Answer

In which phase of the Terraform workflow do you write configuration files based on the scope defined by your organization?

A. Author

B. Initialize

C. Plan

D Scope

Module Review

01 Explain the Terraform workflow.

02 Create basic configuration files within Terraform.

03 Explain the purpose of a few Terraform commands.

04 Describe the Terraform Validator tool.

05 Create, update, and destroy Google Cloud resources using Terraform.