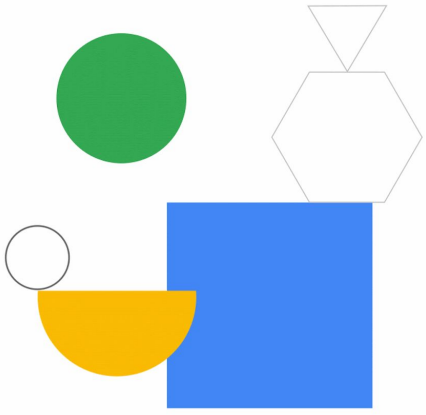
Google Cloud

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.

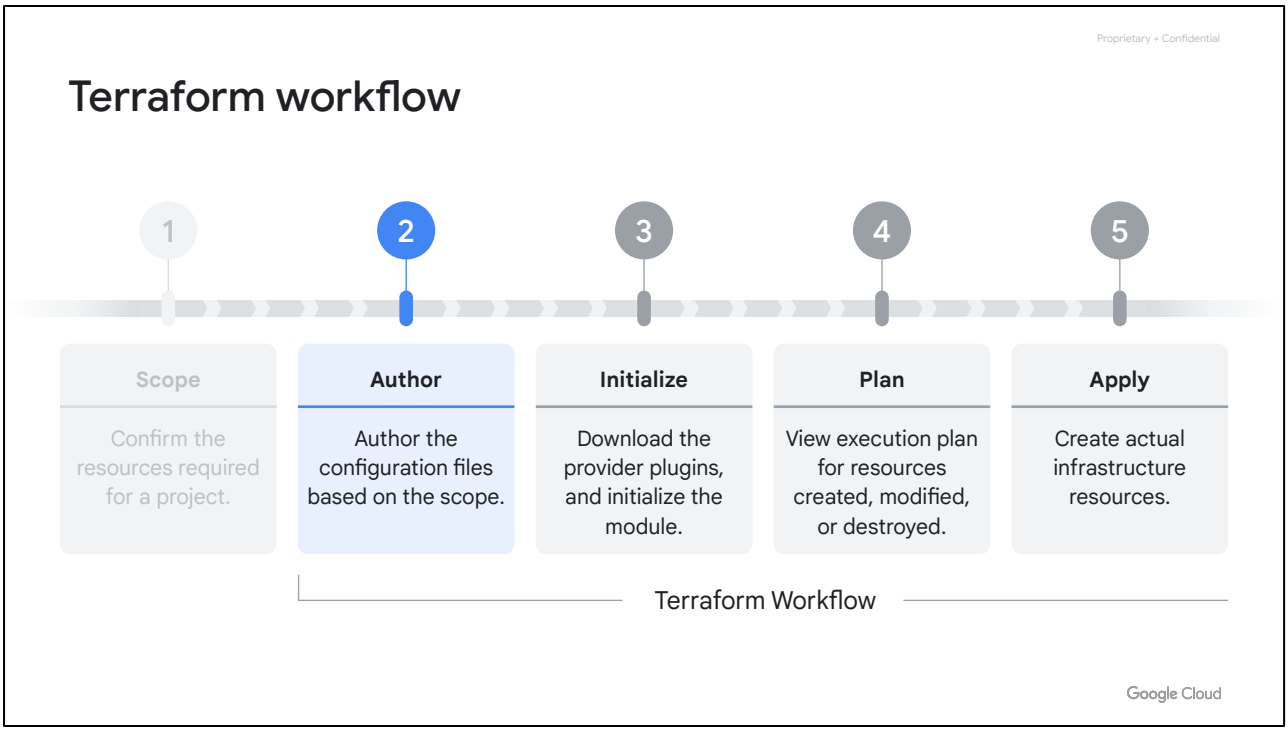
Welcome to the “Terms and Concepts” module. In this module, we introduce you to HashiCorp Language and discuss the terms and concepts involved in authoring a Terraform configuration. We also explore some of the important Terraform commands involved in managing the terraform configuration. Upon completion of this module, you will be able to interpret what each code block means, create basic configuration files within Terraform and be able to explain the purpose of a few few important terraform commands and we will also explore what a validator tool is.

Topics

01 The Author phase

02 Terraform commands

03 Terraform Validator Tool



Throughout this course, we focus on the core Terraform workflow, which is author, initialize, plan and apply. In this module, we will dig deeper to explore how these individual phases fit together in transforming code to cloud resources. Let's start with the author phase, where you write Terraform code in `.tf` files.

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.

Before we write the code, we start by creating directories for Terraform configuration. A Terraform configuration is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. The terraform directory can consist of multiple files and directories.

The configuration written in Terraform language is stored with a .tf extension.

A Terraform configuration consists of a root module and an optional tree for child modules.

- Your root module is the directory in which you'll be running Terraform commands. In that directory, Terraform will look for any .tf files and use them to create a plan and create infrastructure elements. The root module is also referred to as the root configuration file.
- The child modules can be variables, outputs, providers, etc. Child modules are optional, you can have a series of resources and other code constructs within a single root configuration but it is a best practice to logically separate your files as main.tf, providers.tf, variables.tf, outputs.tf, and terraform.tfvars.

Root configurations (root modules) are the working directories from which you run the Terraform CLI.

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
}
```


Now that we know how to create a directory structure for Terraform, let us explore the language used to write Terraform configuration, which is the HashiCorp Configuration language or simply called HCL.

HCL is the configuration language used to create and manage API-based resources, predominantly in the cloud. Resources are infrastructure objects such as virtual machines, storage buckets, containers, or networks. Terraform uses the HCL to define resources in your Google Cloud environment, create dependencies with those resources, and also define the data to be fetched.

Note that despite a few commonalities with programming language, HCL is a configuration language, and not a programming language. It is a JSON-based variant that is human and machine friendly. It is the simplicity of HCL that makes Terraform accessible to developers.

HCL includes a limited set of primitives such as variables, resources, outputs, and modules. It does not include any traditional statements or control loops. The logic is expressed through assignments, count, and interpolation functions.

The generic structure of a Terraform code block is as shown on the slide. The code is structured and organized as blocks and we will explore the syntax in detail in the next slides.

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
}
```


A simple syntax of the Terraform language along with an example is shown on the slide. Let’s explore this HCL syntax in detail.

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
}
```


Blocks are lines of code that belong to a certain type. Examples include resource, variable, and output. A block can be simple or nested to include another block type.

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
}
```


Arguments are part of a block and used to allocate a value to a name. Some blocks have mandatory arguments, while others are optional.

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
}
```


Identifiers are names of an argument, block type, or any Terraform-specific constructs. Identifiers can include letters, underscores, hyphens, and digits, but cannot start with a digit.

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
}
```


Expressions can be used to assign a value to an identifier within a code block. These expressions can be simple or complex.

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
}
```


Comment syntax start with # for a single-line comment.

Use // for a single-line comment, and
/* and */ for multi-line comments.

Remember, HCL is declarative by nature, which means you define the end state of an infrastructure. Therefore the order of the blocks or files does not matter.

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
}
```


Resources are code block type that define the infrastructure components that you would like to define within Terraform.
The resource is identified by the keyword **`resource`**, followed by the resource type and a custom name. The resource type depends on the provider defined in your configuration. Inside the curly brackets are the resource arguments, where you specify the inputs needed for the configuration. Terraform uses the resource type and the resource name to identity an infrastructure element

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"
}
```


Assuming we have Google cloud defined as the provider, shown on slide is an example of Google Cloud Storage Bucket.

- The keyword resource is used to identify the block as the cloud infrastructure component.
- The resource type is `google_storage_bucket` , which is used by terraform to identify the Google Cloud resource. This is a terraform specific terminology and the convention cannot be customized. The resource type vary based on the provider defined.
- The resource name is `example-bucket` . Terraform uses the resource type and the resource name together as an identifier for the resource.

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"
    }
  }
}
```


Shown on slide are examples of 2 different resource blocks, one for a `cloud_storage_bucket` and other for the `google_compute_instance` .
The arguments differ based on the resource type being defined.

- For the `google_storage_bucket` resource, you only need to specify the **name** and **location** to successfully create the resource.
- For the `google_compute_instance` resource, you need to specify the **name**, **machine_type**, and **network_interface.** Zone and tags are optional.

Pro tip: You can use separate files (for example, a file for your instances, storage buckets, and datasets) if you have lengthy resource configurations.

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"
}
```


Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure.

In the providers.tf file, you specify the Terraform block that includes the provider definition you will use. The Terraform {} block is recommended so that Terraform knows which provider to download from the Terraform Registry.

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 configurations belong in the root module of a Terraform configuration.

Shown on the slide is the provider block. The source argument provides the global source address for the provider you intend to use. In the above example, the source argument is assigned to the terraform registry URL hashicorp/google, which is short for registry.terraform.io/hashicorp/google.

The name google is the local name of the provider to be configured. To ensure that the local name is configured correctly, this provider must be included in the required provider block. The arguments such as project and region can be declared within the provider block and are specific to the google provider.

For specifications on configuration arguments and versioned documentation of Google Cloud provider, refer to the Terraform Registry.
Note: When a provider block is not included within a Terraform configuration, Terraform assumes an empty default configuration.

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
}
```


You can also assign a version to each provider defined in the required_providers block. The version argument is optional, but recommended. It is used to constrain the provider to a specific version or a range of versions in order to prevent the download of a new provider that may possibly contain breaking changes. If the version isn't specified, Terraform will automatically download the most recent provider during initialization.

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	variables.tf	terraform.tfvars
Declare storage bucket	Parameterize resource attribute	Define values at run time or in a file
`location`	`variable location { .. }`	`location= "US"` `storage_class="STANDARD"`
`unique_name`		
`storage_class`		
`versioning`		

CLI

```
$terraform apply

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


Variables in Terraform are a way to parameterize your configuration. Input variables serve as parameters for Terraform, allowing easy customization and sharing without having to alter its source code. Once a variable is defined, there are different ways to set its values at runtime: environment variables, CLI options, or key/value files. It is a way to define centrally controllable values to the resource attributes. Using variables you can easily separate attributes from deployment plans. When you define a resource attribute as a variable, it provides the flexibility to define the value at run time or edit in a file with .tfvars extension.

In the above example, the main.tf has the declaration of Google storage bucket. The `location` attribute has been parameterized by declaring it as variables in the variables.tf file. By parameterizing the attribute you can define the values of these variables at run time when running terraform apply or in a file with a called terraform.tfvars extension.

We will explore variables in more detail in the next module.

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/..
```


Outputs.tf is the file to hold your output values from your resources. Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration. Output values are a way to expose some of that information if needed.

Some attributes of a resource are computed upon its creation. For example, a self link of the resource or the URL of a bucket is generated upon bucket creation. These computed attributes might be required to be used for accessing the bucket or upload objects into bucket. You can output this information and make it available to the user by declaring it as a output value. The label immediately after the output keyword is the **name**, which must be a valid identifier. In a root module, this name is displayed to the user; in a child module, it can be used to access the output's value.

- The **value** argument takes an expression whose result is to be returned to the user.

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": [
        {
```


Terraform saves the state of resources it manages in a state file. State is used to manage infrastructure after creation. By default, the state file is stored locally, but for team or bigger projects can also be stored remotely.
Do not modify or touch this file; it is created and updated automatically. We will explore more about states in module 5.

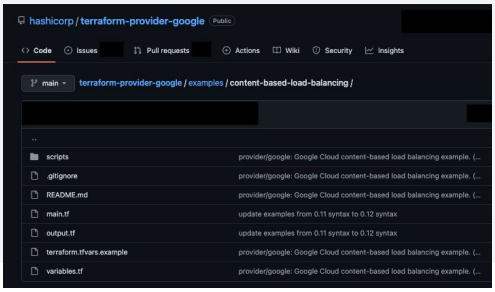
Modules

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



A Terraform module is a set of Terraform configuration files in a single directory. Even a simple configuration that consists of a single directory with **one or more .tf** files is a module.

It is the primary method for code reuse in Terraform. The modules are reused by specifying the source from which the code can be fetched.
There are 2 kinds of sources:

- Local: Source within your directory
- Remote: Source outside your directory.

You can use a upstream module from the HashiCorp module registry or create your own.

Topics

01 The Author phase

02 Terraform commands

03 Terraform Validator tool

Once Terraform is installed on your machine, you use these commands at your root module to interact with Terraform.

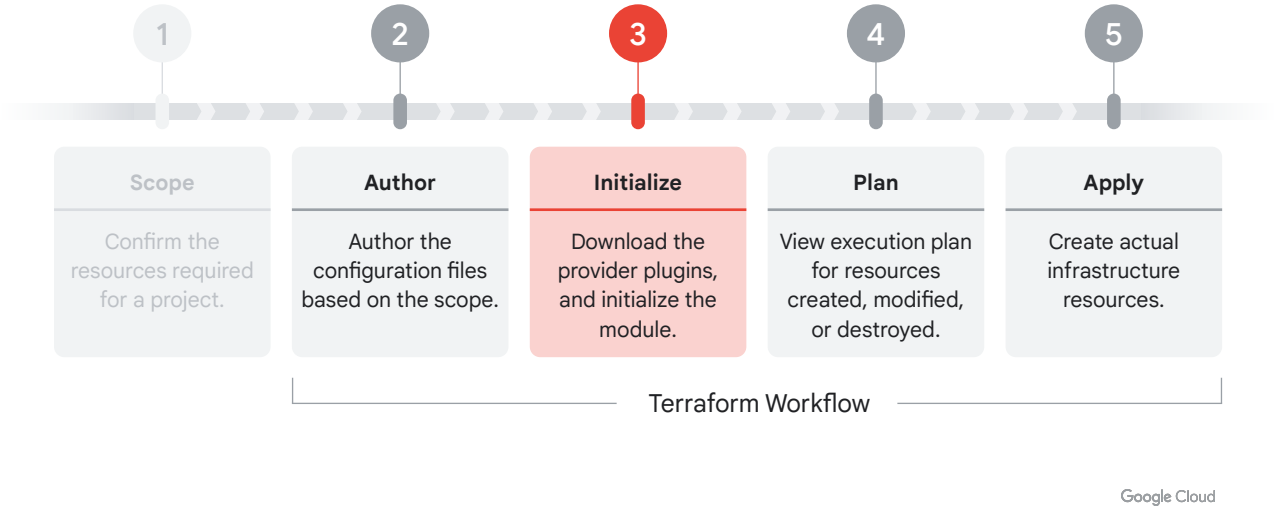
Terraform commands

`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

After Terraform is installed on your machine, you use a few commands at your root module to interact with Terraform. Some of the important commands that we discuss in this section are `terraform init`, `terraform plan`, `terraform apply`, `terraform destroy`, and `terraform fmt` .

- `terraform init` is used to initialize the provider with plugin.
- `terraform plan` provides a preview the resources that will be created after terraform apply.
- `terraform apply` creates read infrastructure resources.
- `terraform destroy` destroys infrastructure resources.
- `terraform fmt` auto formats to match canonical conventions.

Initialize Terraform using `terraform init`



It is in the initialize phase that the `terraform init` command is run. The first command to run for a new configuration—or after checking out an existing configuration from version control—is `terraform init`. The `terraform init` command makes sure that the Google provider plugin is downloaded and installed in a subdirectory of the current working directory, along with various other bookkeeping files.

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"
    }
  }
}
```

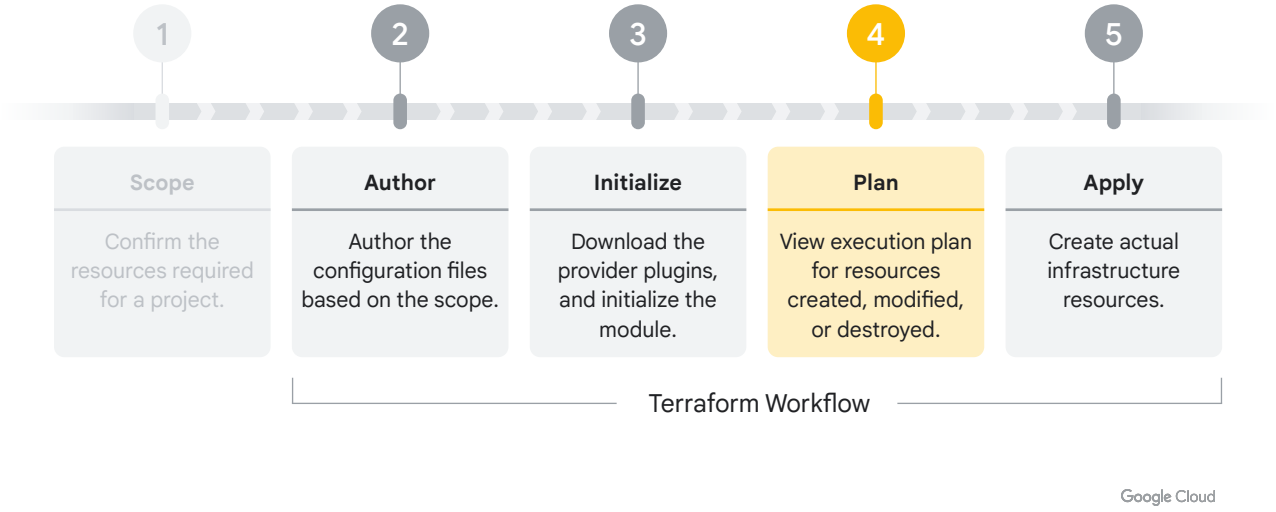
```
$ terraform init
Initializing the backend...

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!
```


When writing the Terraform configuration, the provider block includes the source attribute that specifies the location from which the provider plugins need to be downloaded. Terraform uses a plugin-based architecture to support the numerous infrastructure and service providers available. Each "provider" is its own encapsulated binary that is distributed separately from Terraform itself. The `terraform init` command will automatically download and install any provider binary for the providers to use within the configuration, which in this case is the Google provider.

After you execute `terraform init,` a hidden directory called `.terraform` is created under the current working directory. You will see an "`Initializing provider plugins`" message when running the command, which indicates that Terraform will find the latest plugin from the URL and downloads the associated files. In the output of this command, you can also see the provider version that Terraform has installed. In this case, it is version 4.21.

Preview resource action using `terraform plan`



`Terraform plan` - creates an execution plan that details all the resources will be created, modified, or destroyed upon executing the `terraform apply` command. Run `terraform plan` to verify creation process.

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.
```

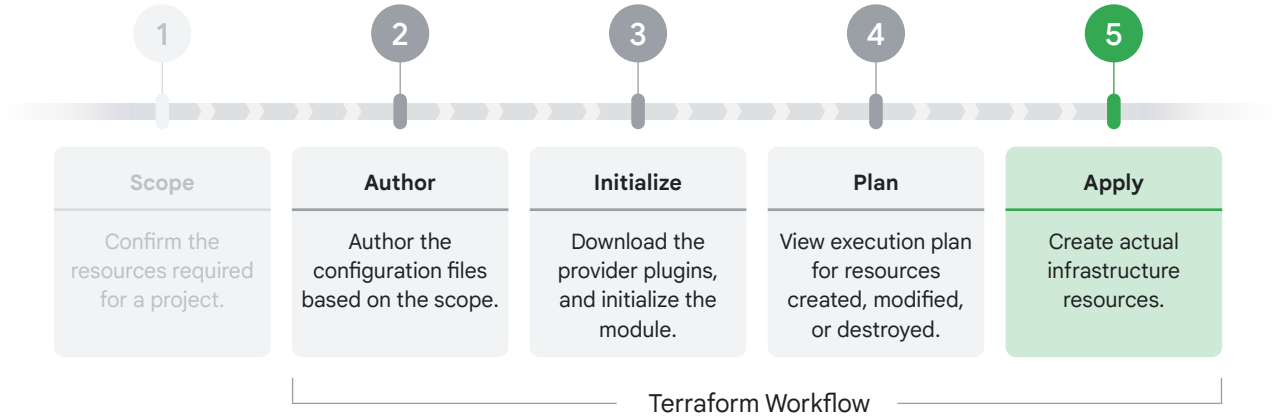

By default, creating a plan consists of:

- Reading the current state of any already existing remote objects to make sure that the Terraform state is up to date.
- Comparing the current configuration to the prior state and noting any differences.
- As you change Terraform configurations, Terraform builds an execution plan that only modifies what is necessary to reach your desired state.

This command does not actually create or change any infrastructure resources but instead provides you with an opportunity to preview the changes that you are making to the infrastructure before applying them. For example, you might be run this command before committing a change to version control in order to be sure that it will behave as expected.

You can use the optional `-out=FILE` option to save the generated plan to a file on disk, which you can later execute by passing the file to terraform apply as an extra argument.

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



Terraform apply executes the actions proposed in a Terraform plan, creates the resources, and establishes the dependencies.

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.
```


The sign next to the resource and arguments, indicates the action performed on the resource:

- The plus next to the resource means that Terraform will create this resource. Terraform also shows the attributes that will be set.
- Minus slash plus means that Terraform will destroy and recreate the resource, rather than updating it in-place.
- The tilde means that Terraform will update the resource in-place.
- The minus indicates that the instance and the network will be destroyed.

As with `terraform apply`, Terraform shows its execution plan and waits for approval before making any changes.

- If the plan was created successfully, Terraform will now pause and wait for approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here with no changes made to your infrastructure.
- If terraform apply failed with an error, read the error message and fix the error that occurred.

Just like with `terraform plan`, Terraform determines the order in which things must be destroyed. For example, Google Cloud won't allow a VPC network to be deleted if there are resources still in it, so Terraform waits until the instance is destroyed before destroying the network.

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"
       }
     }..
}
```


Some of the code conventions to be followed in HCL are as follows:

- Separate the meta arguments from the other arguments by placing it first or last in the code with a blank line.
- Indent your arguments with two space from block definition
- When two or more arguments are defined in a given block align the values at the equal sign
- When a block includes a nested block place them below all the arguments.
- When your code includes multiple blocks, separate them with a black line for readability.

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"
        }
      }..
```

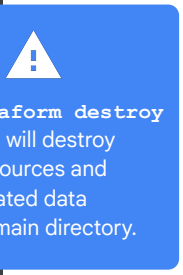

Terraform fmt can enforce code convention best practices.

`Terraform fmt` - automatically updates the configuration for readability and consistency. Running `terraform fmt` on your modules and code automatically applies all formatting rules and recommended styles.
Use `terraform fmt` as it automatically maintains consistent formatting for you so you don't have to manually change configuration to ensure it meets the standards.

We will explore more best practices as we move further into the course.

`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.
```



Resources can be destroyed using the `terraform destroy` command, which is similar to `terraform apply` but it behaves as if all of the resources have been removed from the configuration.

- While you will typically not want to destroy long-lived objects in a production environment. Terraform is sometimes used to manage ephemeral infrastructure for development purposes, in which case you can use `terraform destroy` to conveniently clean up all of those temporary objects once you are finished with your work. You can also destroy specific resources by specifying a target in the command.

Note: Destroying your infrastructure is a rare event in production environments. But if you're using Terraform to spin up multiple environments such as development, testing, and staging, then destroying is often a useful action. `terraform destroy` will destroy any resource and if the resource stores any data then the associated data too. For example, if there is data in a bucket, be careful when running `terraform destroy` as that data cannot be recovered.

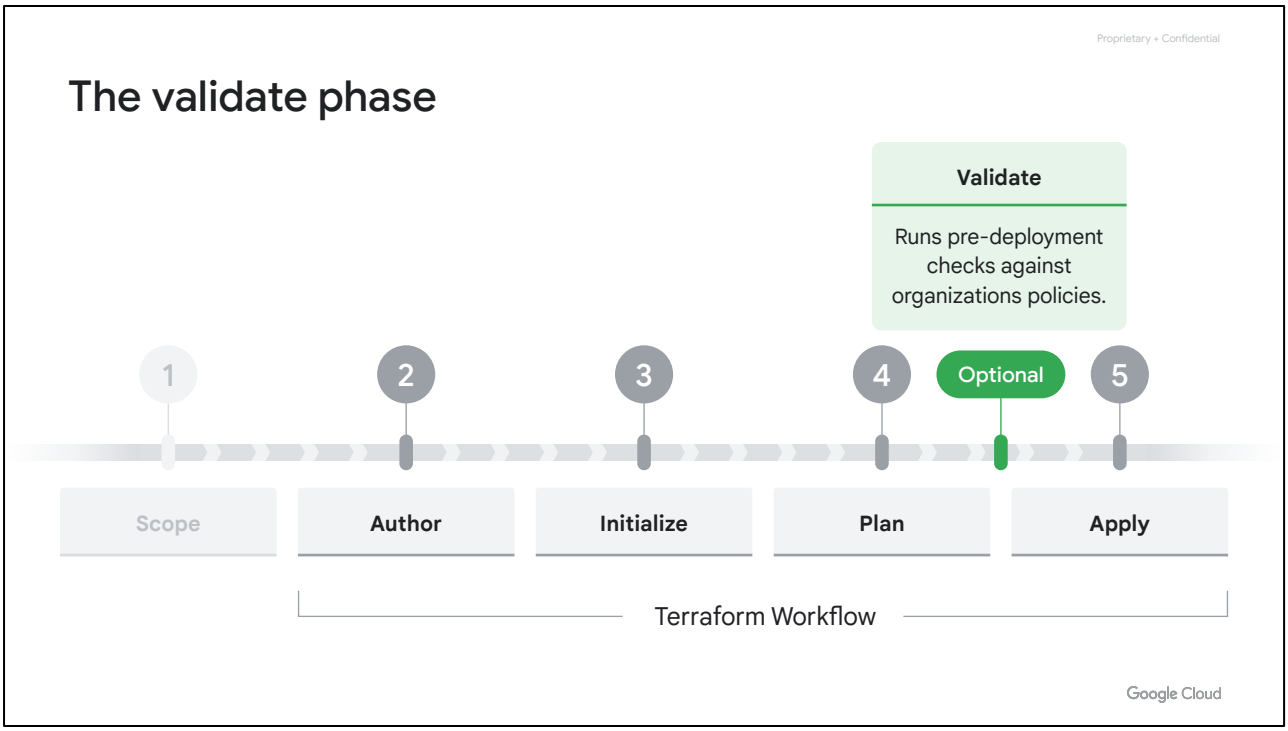
Topics

01 The Author phase

02 Terraform commands

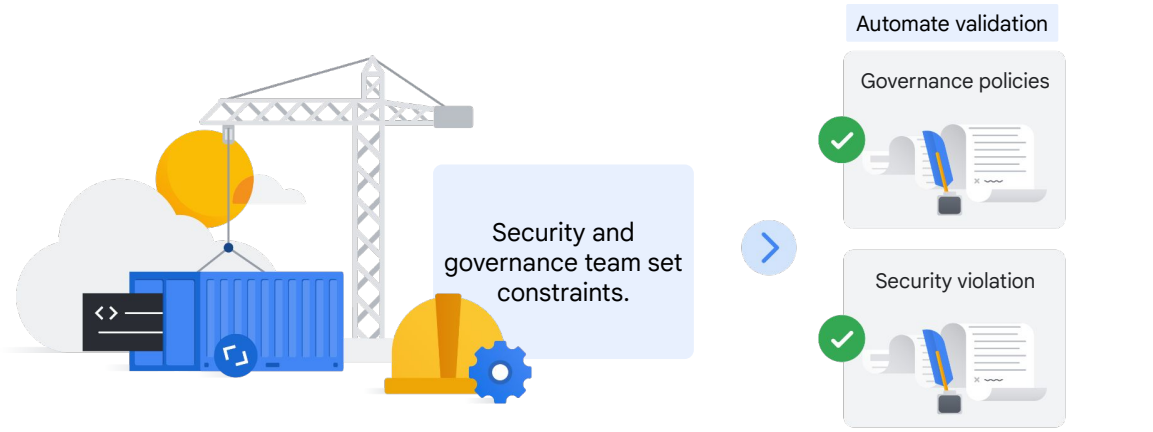
03 Terraform Validator Tool

So far we spoke about the standard Terraform workflow. Terraform workflow includes an optional phase called Terraform validate. This phase is between the plan and apply phase.



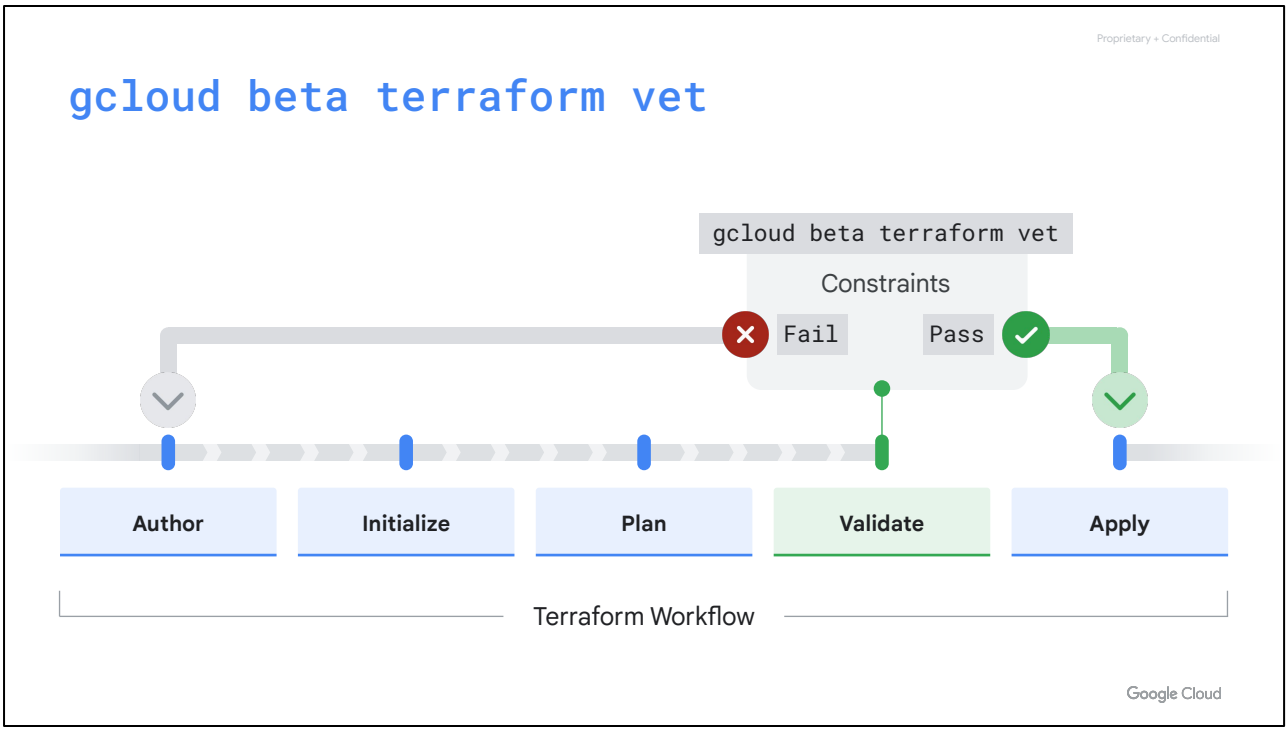
Post the Terraform plan stage, you can optionally include a validate stage that runs pre-deployment checks against organizations policies. Terraform validator is a tool for enforcing policy compliance as part of an infrastructure CI/CD pipeline. This is extremely useful in an infrastructure-as-code environment as it helps mitigate the configuration errors that can cause security and governance violations. The terraform validator is run by executing the **`gcloud beta terraform vet`** command.

Why use the Terraform validator tool?



Although, Terraform validator is not a mandatory tool to have, let's explore what value Terraform validator adds to an IaC environment. Many organizations have compliance and governance policies in place that have to be adhered to. If you were wondering a way to automate the validation piece in the deployment pipeline too. Terraform validator is the tool for you. Businesses are shifting towards infrastructure-as-code, and with that change comes a concern that configuration errors can cause security and governance violations. For example, the governance team would want to ensure creation of resources are only allowed in certain regions in order to adhere to data residency laws. To address this, the security and governance team need to be able to set up guardrails that make sure everyone in their organization follows security best practices. These guardrails are in the form of constraints.

Constraints define your organization's source of truth for security and governance requirements. The constraints must be compatible with tools across every stage of the application lifecycle, from development, to deployment, and even to an audit of deployed resources.



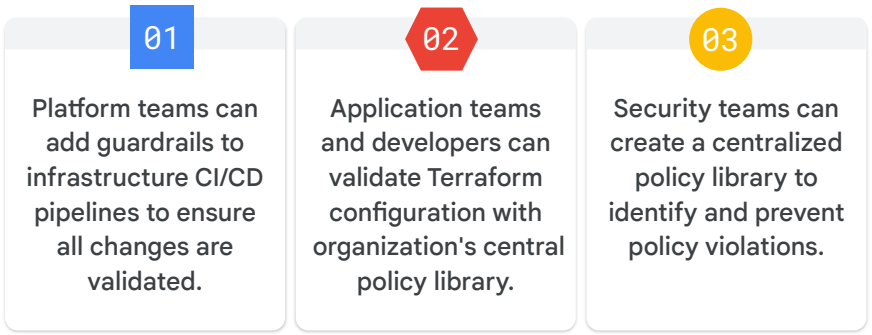
`gcloud beta terraform vet` is a tool for enforcing policy compliance as part of an infrastructure CI/CD pipeline. When you run this tool, gcloud beta terraform vet retrieves project data with Google Cloud APIs that are necessary for accurate validation of your plan. You can use gcloud beta terraform vet to detect policy violations and provide warnings or halt deployments before they reach production. The same set of constraints that you use with gcloud beta terraform vet can also be used with any other tool that supports the same framework.

With `gcloud beta terraform vet` you can:

- Enforce your organization's policy at any stage of application development
- Remove manual errors by automating policy validation
- Reduce learning time by using a single paradigm for all policy management

Note that this tool is completely different from the command "`terraform validate`". We have not covered this command in this course. The command '`terraform validate`' is used for testing the syntax and structure of your configuration without deploying any resources. But the `gcloud beta terraform vet` command is used to ensure that the configuration adheres to the set of constraints. These constraints automate the enforcement of the organization policies.

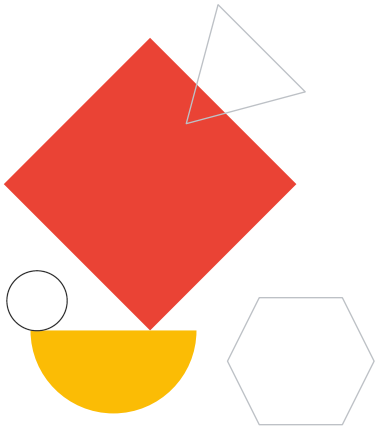
Terraform Validator uses



- Platform teams can easily add guardrails to infrastructure CI/CD pipelines (between the plan & apply stages) to ensure all requests for infrastructure are validated before deployment to the cloud. This limits platform team involvement by providing failure messages to end users during their pre-deployment checks which tell them which policies they have violated.
- Application teams and developers can validate their Terraform configurations against the organization's central policy library to identify misconfigurations early in the development process. Before submitting to a CI/CD pipeline, you can easily ensure your Terraform configurations are in compliance with your organization's policies, thus saving time and effort.
- Security teams can create a centralized policy library that is used by all teams across the organization to identify and prevent policy violations. Depending on how your organization is structured, the security team (or other trusted teams) can add the necessary policies according to the company's needs or compliance requirements.

Demo

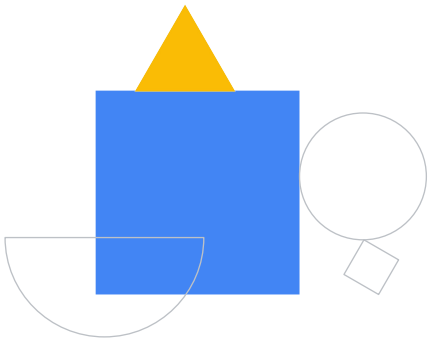
Demonstrate Terraform Workflow



This demonstration will help you become familiar with the Terraform configuration to create a Compute Engine instance. We will show you how to create configuration files and use the Terraform CLI to execute a few Terraform commands such as terraform init, terraform plan, terraform apply, and terraform destroy.

Lab

Infrastructure as Code with Terraform



In this lab you will learn how to perform the following tasks:

- Verify Terraform installation
- Define Google Cloud as the provider
- Create, change, and destroy Google Cloud resources by using 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

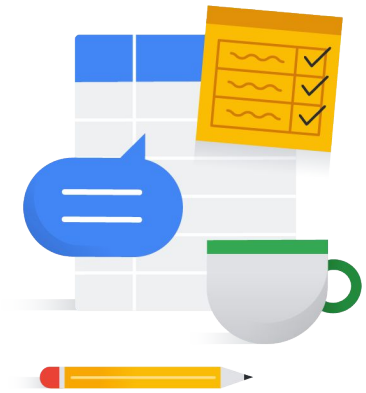
01 Explain the Terraform workflow.

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.



This module described terms and concepts relating to each phase of the Terraform workflow. You learned how to create basic configuration files within Terraform, and how to describe a Terraform provider. This module also explained the purpose of a few important Terraform commands. In addition, you learned about an optional phase in the Terraform workflow called validate. You also learned how to create, update, and destroy Google Cloud resources.

Check out the next module to learn more about writing infrastructure code.