
cloudthat
move up.


Terraform

About CloudThat

9+
Years in Business

350k+
Professional Trained

100+
Corporates Trained

100+
Projects Delivered

350+
Cloud Certifications

Leader in Training and Consulting on Cloud, Security, AI/ML, IoT and DevOps

Trained over 350k+ professionals across technologies and geographies

Proven track record of training delivery for all stages of employee lifecycle

Strong team of 125+ certified cloud experts with industry consulting experience

Robust consulting division brings industry prospective to training delivery

Gold Microsoft Partner
Microsoft

Microsoft Partner
2020 Partner of the Year Finalist
Learning Award

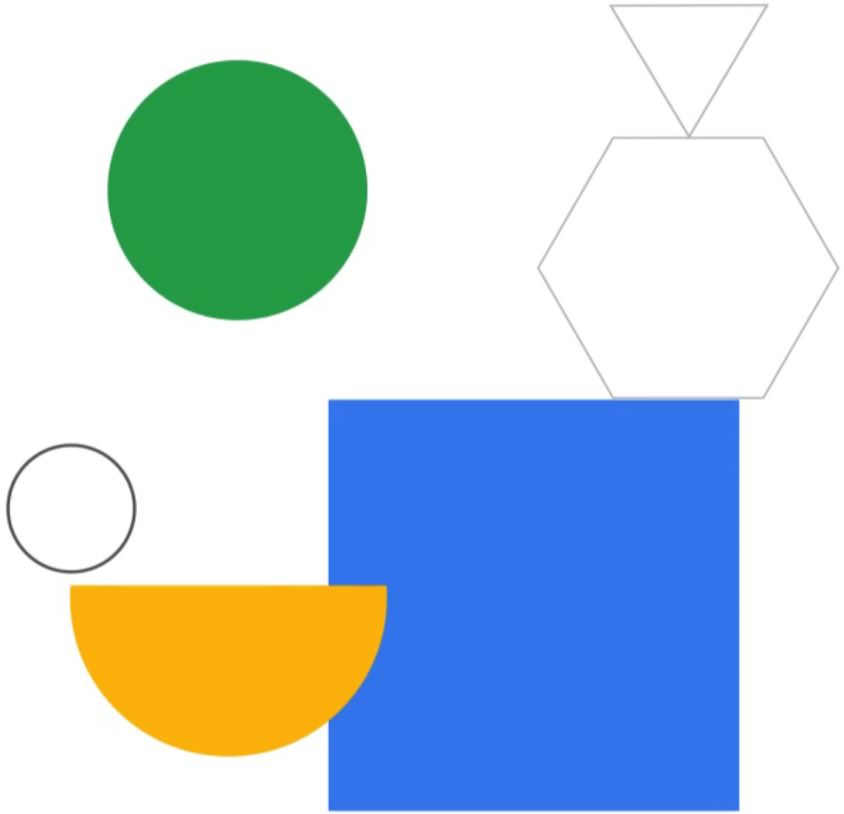
aws partner network
Advanced Consulting Partner

Google Cloud Partner

cloudthat

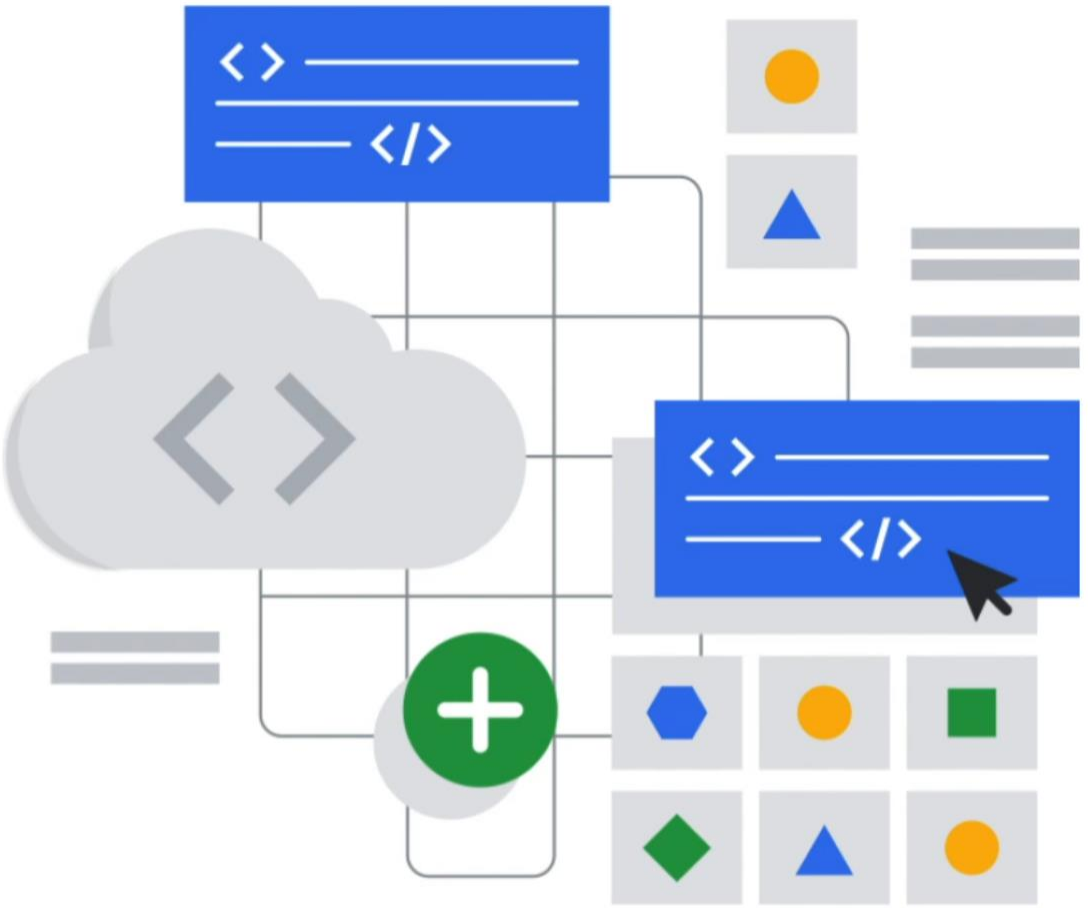
Google Cloud

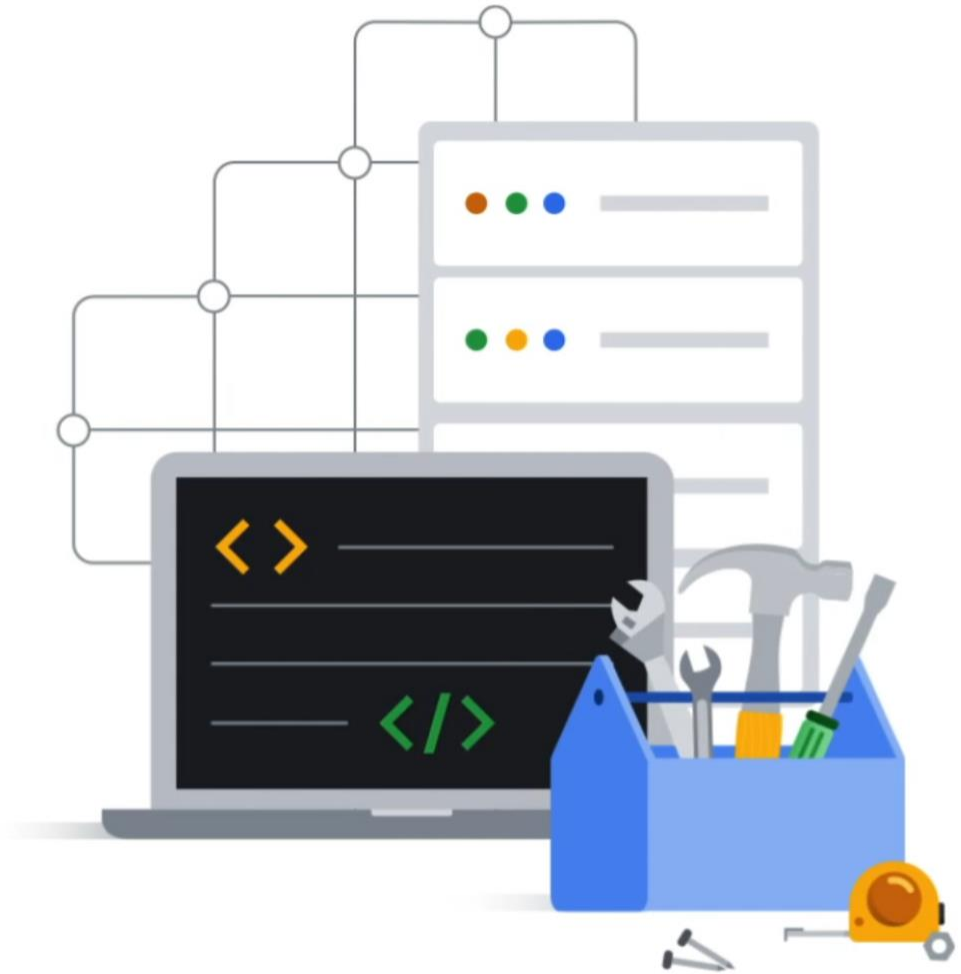
Introduction to Terraform for Google Cloud



What is infrastructure as code (IaC)?

Instead of clicking around a web UI or using SSH to connect to a server and manually executing commands, **with IaC you can write code in files to define, provision, and manage your infrastructure.**





DevOps

Emphasizes the collaboration and communication of both software developers and IT operations teams

Automate the process of software delivery and infrastructure changes

Problems that IaC can solve

Inability to scale rapidly

Requires rapid scaling of IT infrastructure

Operational bottlenecks

Challenge of managing infrastructure consistently in scale

Disconnected feedback loops

Communication gap between software and IT teams

High manual errors

Increased scale leads to increased human errors

Benefits of IaC

Declarative	Specify the desired state of infrastructure, not updates.
Code management	Commit, version, trace, and collaborate, just like source code.
Auditable	Compare infrastructure between desired state and current state.
Portable	Build reusable modules across an organization.

Provisioning versus configuration

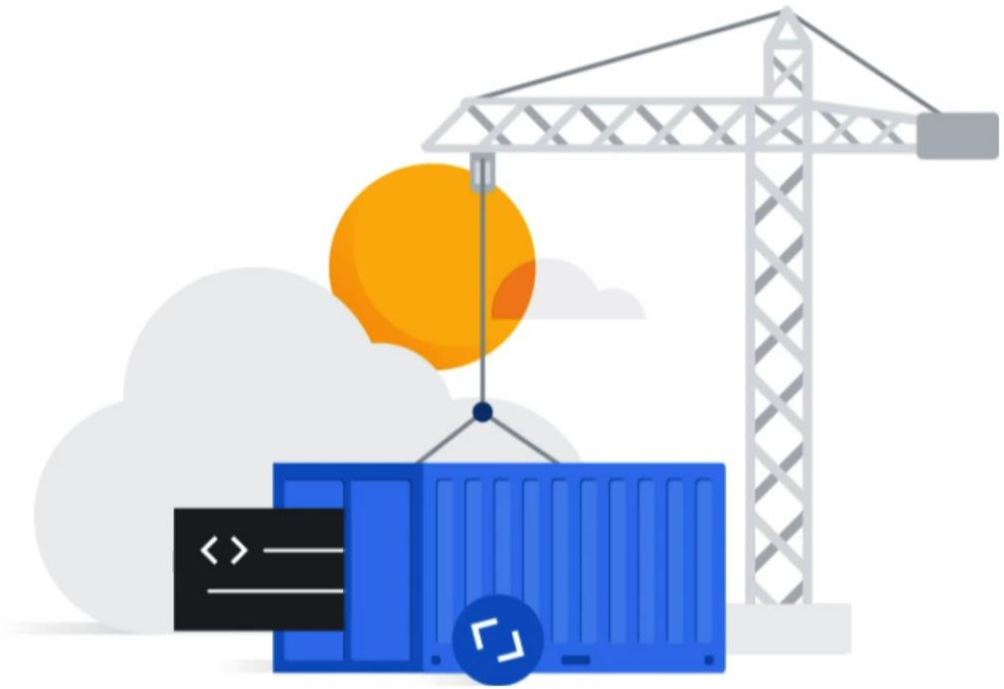
Infrastructure as code

- Used for provisioning and managing cloud resources.
- Example: Creating and provisioning a VM instance.
- Referring to frameworks that manipulate Google Cloud APIs to deploy the infrastructure.

Configuration Management

- Used for virtual machine OS-level configuration.
- Example: Configuring the internals of the VMs.
- Referring to package configurations and software maintenance.

Provisioning versus configuration



Infrastructure as a code

- Launch a GKE cluster

Configuration management

- Deploy containers into the GKE cluster

IaC takes a declarative approach to infrastructure

Imperative (command)

Command line

"Give me five servers"

How to create?

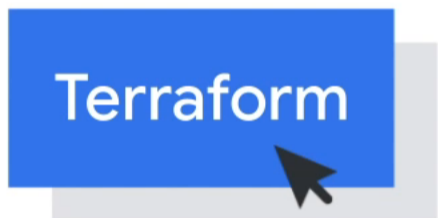
VS.

Declarative (statement)

YAML

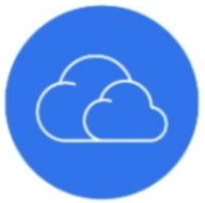
"I should have five servers"

What to create?



Terraform is an open source infrastructure as code tool created by HashiCorp that lets you provision Google Cloud resources with declarative configuration files

Terraform features



Multi-cloud and multi-API

Open core with enterprise support

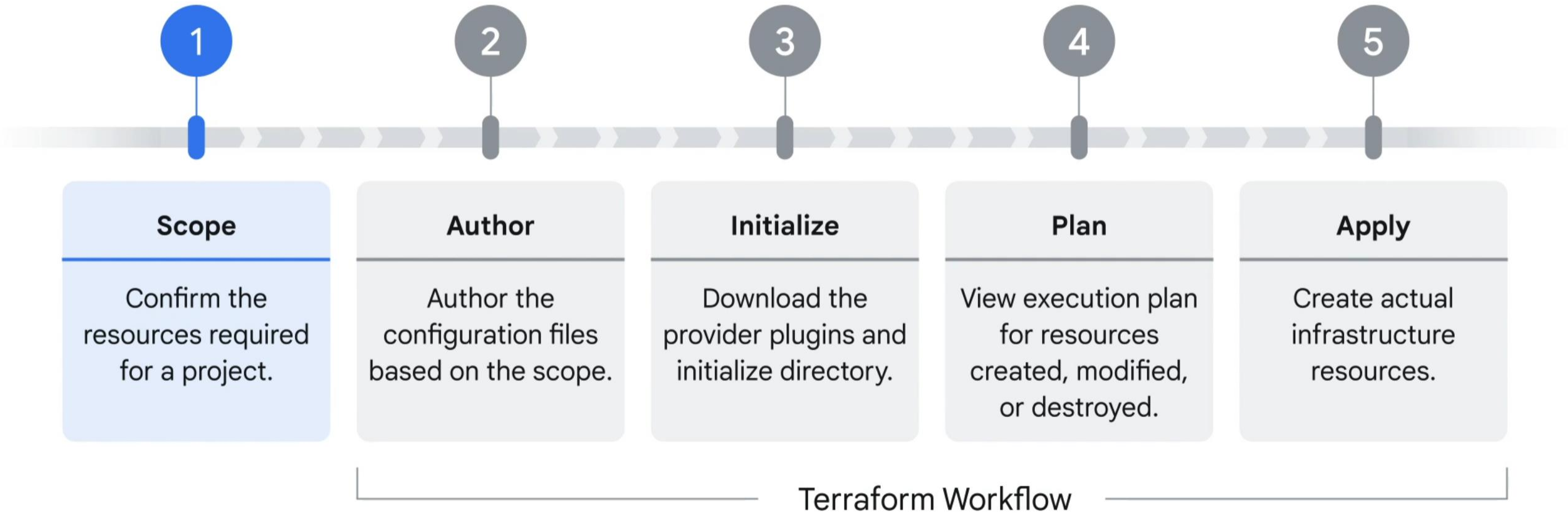
Large community

Infrastructure provisioning

Terraform for Google Cloud

- Provision resources
- Create resource dependencies
- Standardize configurations
- Validate inputs to resource arguments

IaC configuration workflow

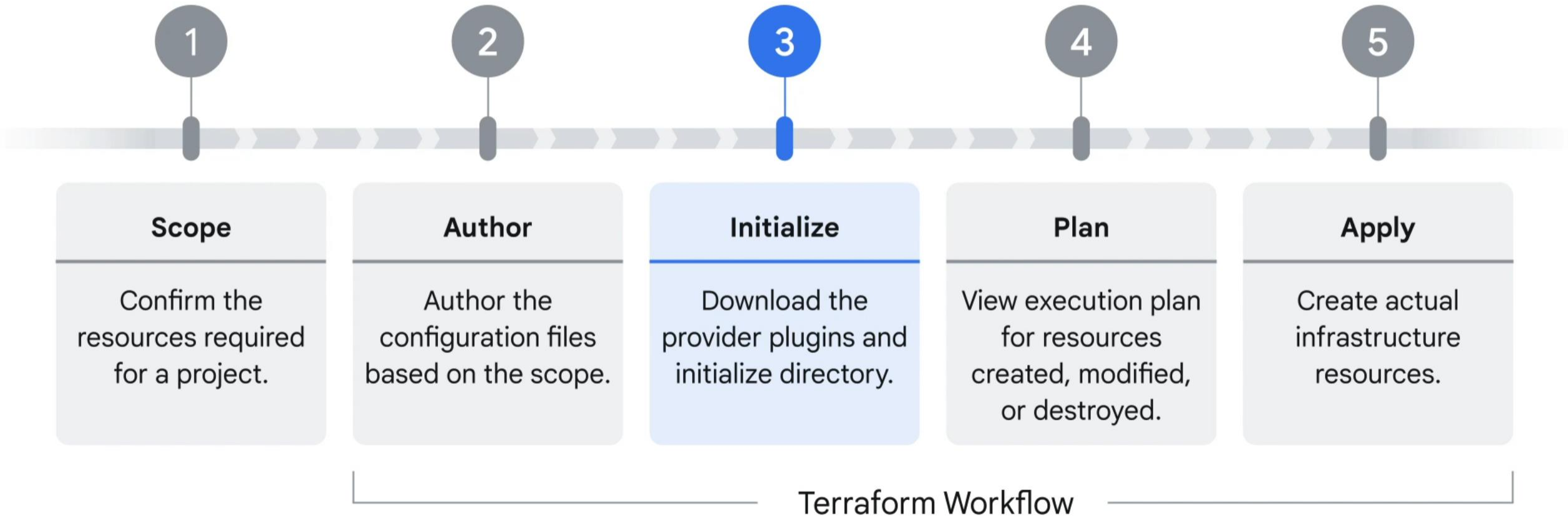


IaC configuration workflow

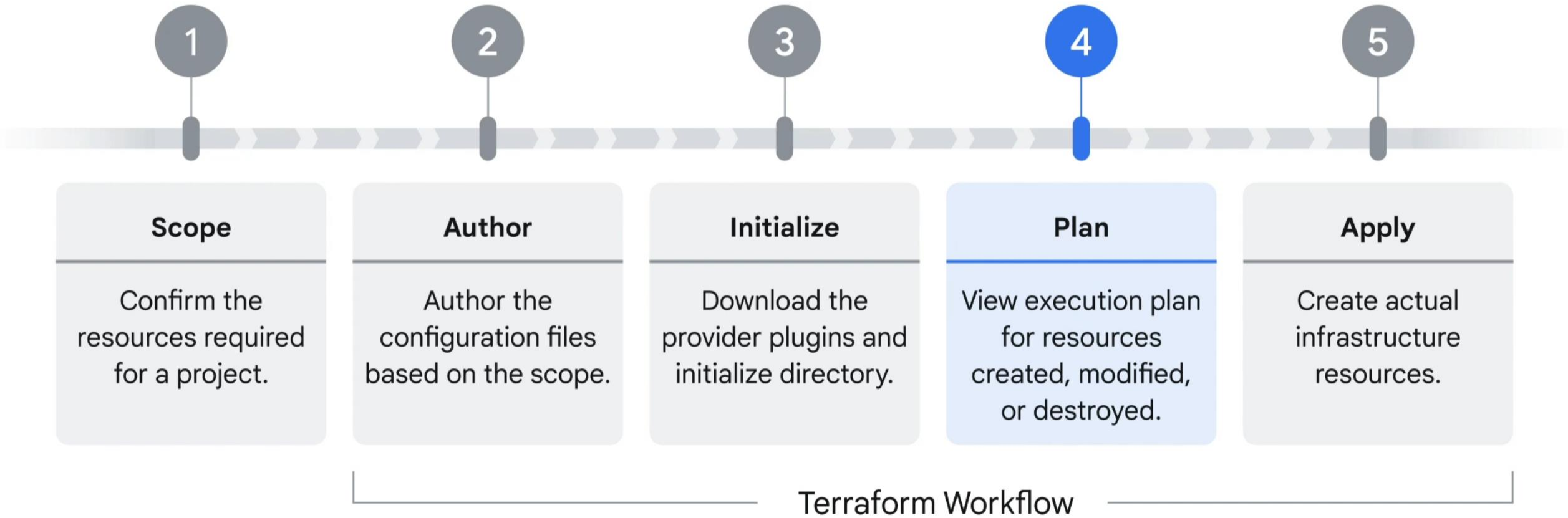
Scope	Author	Initialize	Plan	Apply
Confirm the resources required for a project.	Author the configuration files based on the scope.	Download the provider plugins and initialize directory.	View execution plan for resources created, modified, or destroyed.	Create actual infrastructure resources.

Terraform Workflow

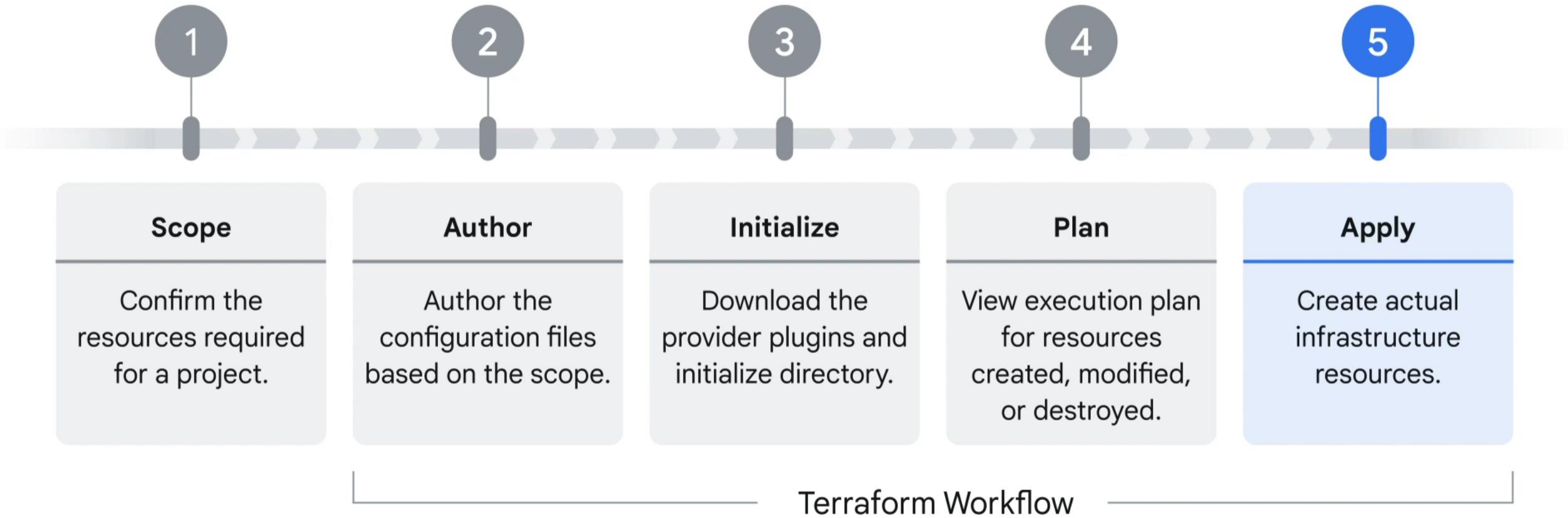
IaC configuration workflow



IaC configuration workflow



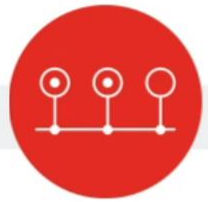
IaC configuration workflow



Terraform use cases

Manage infrastructure

Terraform takes an immutable approach to building and managing infrastructure.



Track changes

Terraform enables you to review the changes before they are applied to the configuration setup.

Automate changes

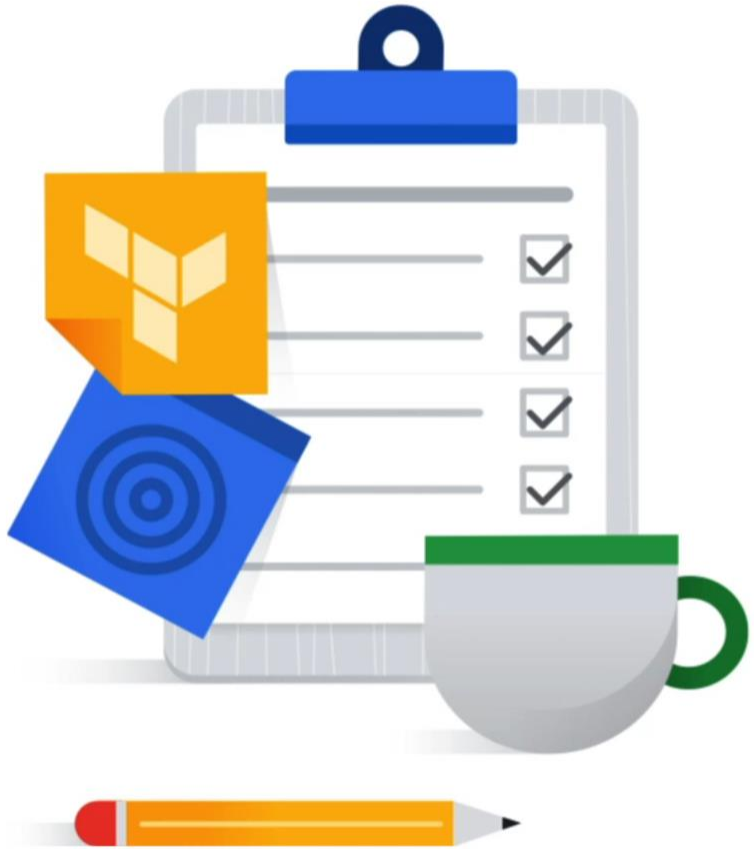
Terraform defines the end state of the infrastructure instead of a series of steps to achieve it.

Standardize the configuration

Terraform uses modules to implement best practices and improve efficiency.

Using Terraform

1. Terraform recognizes configuration files written in .tf file.
2. Terraform generates an execution plan.
3. Terraform uses this plan to create infrastructure.
4. Terraform determines the changes and creates incremental execution plans.



Running Terraform in production

	Managed	Pros	Cons
Terraform Open Source	✕	• Deployed on a local machine or compute resource in the cloud • No license cost • Use public registry within your code	• Does not support concurrent deployments • Only interfaced through CLI
Terraform Cloud	✓	• SaaS based version • Small operational overhead • Comes with three plans • Supports concurrent deployments • Can be accessed through GUI and CLI	• License cost for advanced features
Terraform Enterprise	✓	• Private implementation • Supports concurrent deployments • Secure deployment • Can be accessed through GUI and CLI	• Infrastructure and license costs • Large operational overhead

Running Terraform in production

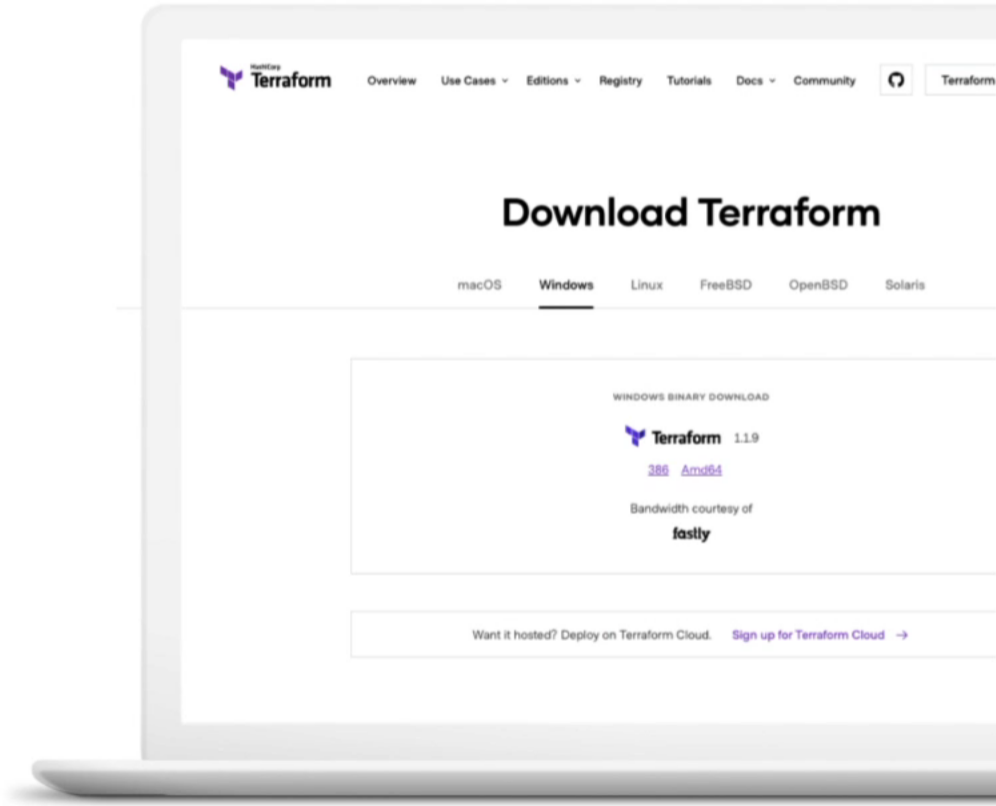
	Managed	Pros	Cons
Terraform Open Source	✗	• Deployed on a local machine or compute resource in the cloud • No license cost • Use public registry within your code	• Does not support concurrent deployments • Only interfaced through CLI
Terraform Cloud	✓	• SaaS based version • Small operational overhead • Comes with three plans • Supports concurrent deployments • Can be accessed through GUI and CLI	• License cost for advanced features
Terraform Enterprise	✓	• Private implementation • Supports concurrent deployments • Secure deployment • Can be accessed through GUI and CLI	• Infrastructure and license costs • Large operational overhead

Running Terraform in production

	Managed	Pros	Cons
Terraform Open Source	✗	• Deployed on a local machine or compute resource in the cloud • No license cost • Use public registry within your code	• Does not support concurrent deployments • Only interfaced through CLI
Terraform Cloud	✓	• SaaS based version • Small operational overhead • Comes with three plans • Supports concurrent deployments • Can be accessed through GUI and CLI	• License cost for advanced features
Terraform Enterprise	✓	• Private implementation • Supports concurrent deployments • Secure deployment • Can be accessed through GUI and CLI	• Infrastructure and license costs • Large operational overhead

Installing Terraform on local machine

1. Download the appropriate package based on your system.
2. Unzip the package. Terraform includes a single binary called `terraform`.
3. Edit the PATH variable to include `terraform`.
4. To verify the installation, enter `terraform -help` in a new terminal.



Authentication for Google Cloud

On your workstation

Authenticate Terraform using Google Cloud SDK. On Cloud Shell, it is pre-authenticated for you.

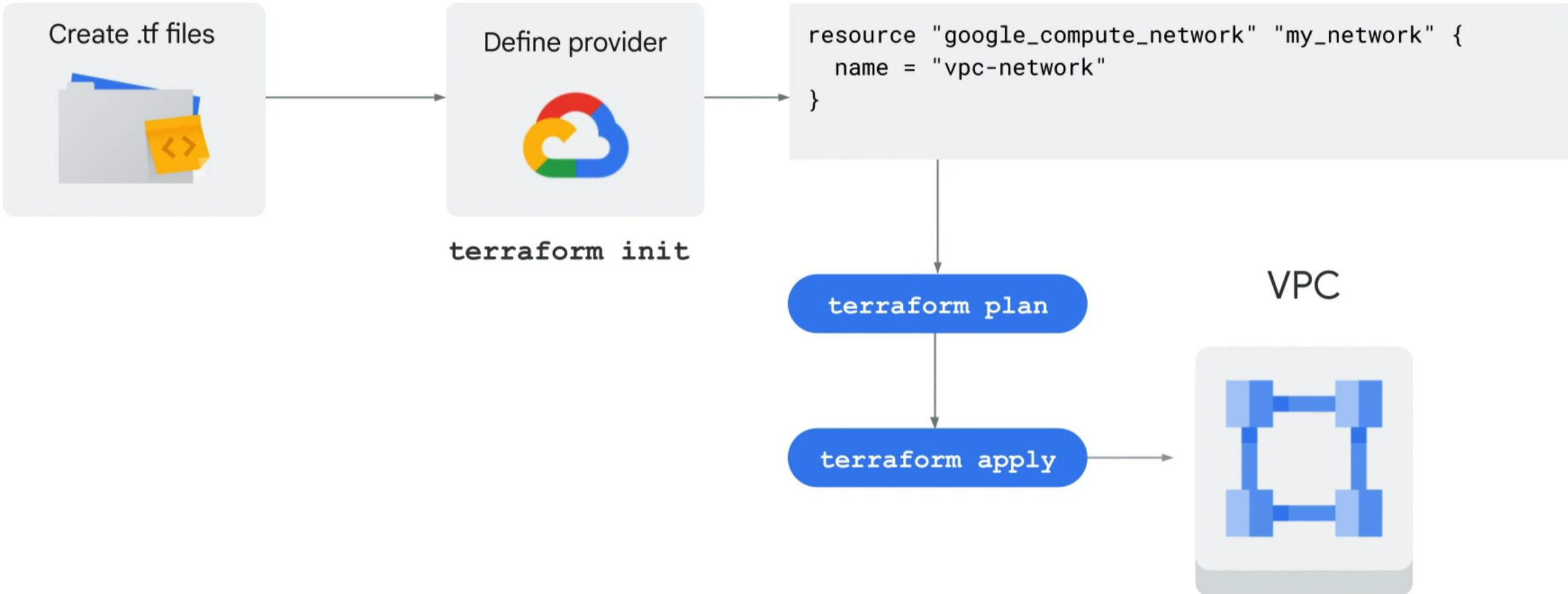
In a VM on Google Cloud

Configure the VM to use a Google Service Account.

Outside Google Cloud

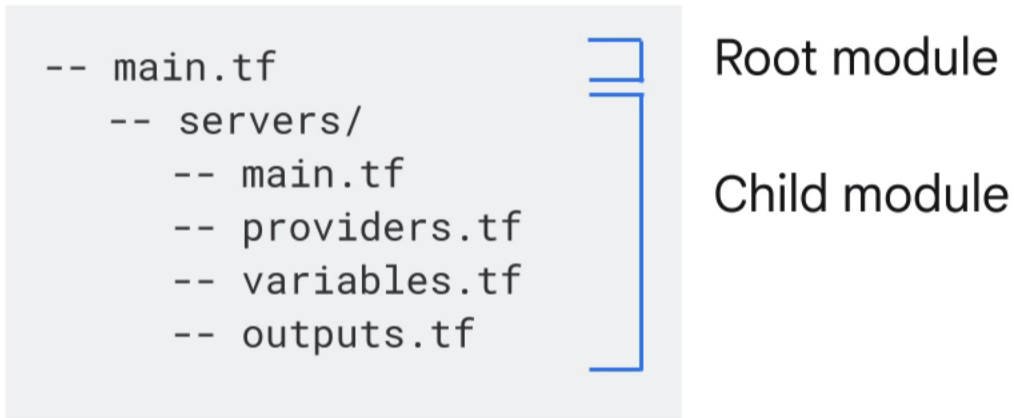
Using workload identity federation, generate a service account key and set environment variables.

Example: Creating a VPC network



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
  - Variable.tf (optional but recommended)
  - Outputs.tf (optional but recommended)
- Terraform commands are run on the working directory.



HashiCorp Configuration Language (HCL)

Syntax

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

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.
- Includes no 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" {
# 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
```

- 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
}
```

Examples of resources

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

- The keyword `resource` is used to identify the block as the cloud infrastructure component.

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

Examples of resources

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

- 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
```

- 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"
    }
  }
}
```

Providers

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

- 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"
}
```

Providers

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

- 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
```

- 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

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

- 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.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

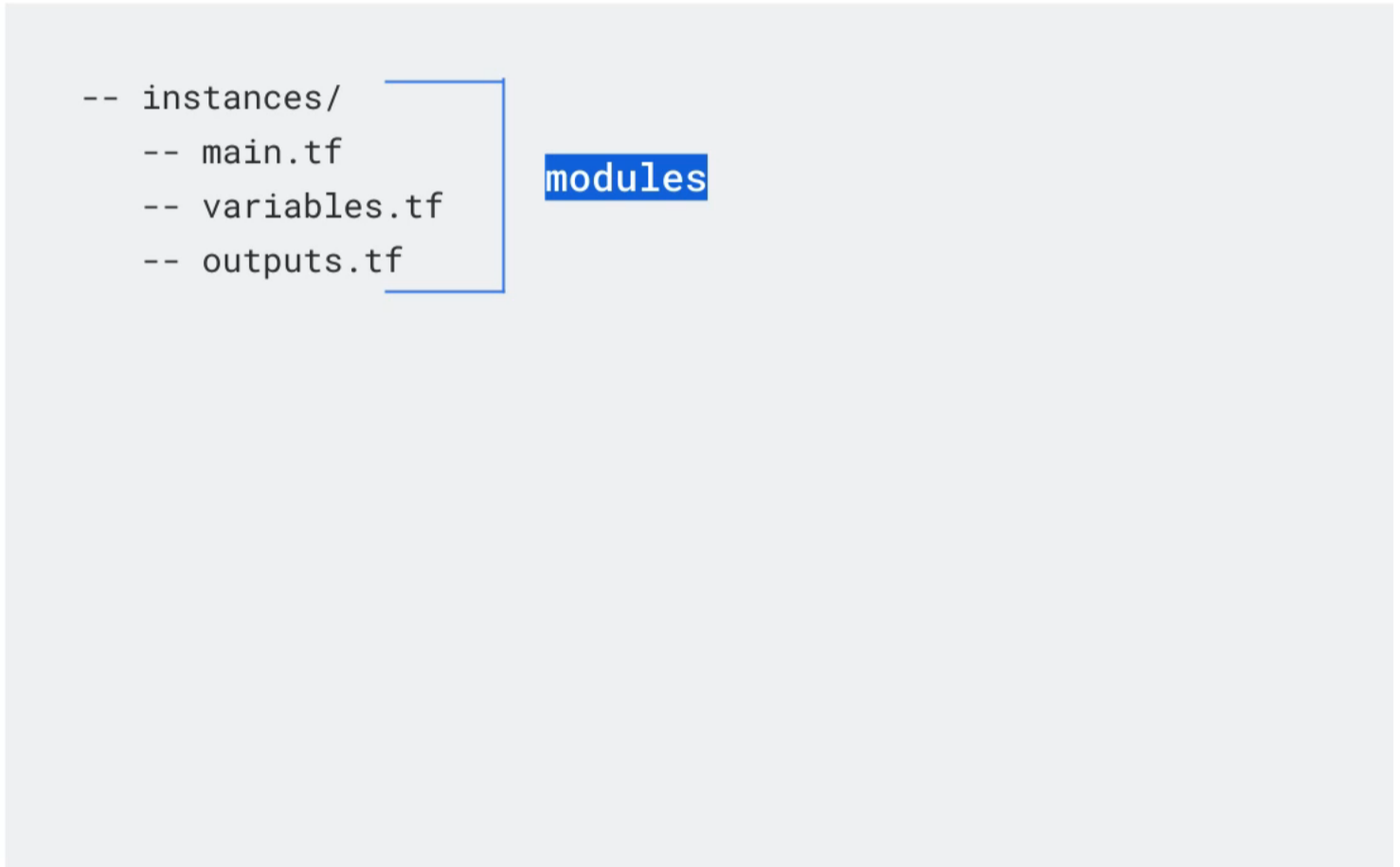
You do not modify this file.


```
{
  "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.

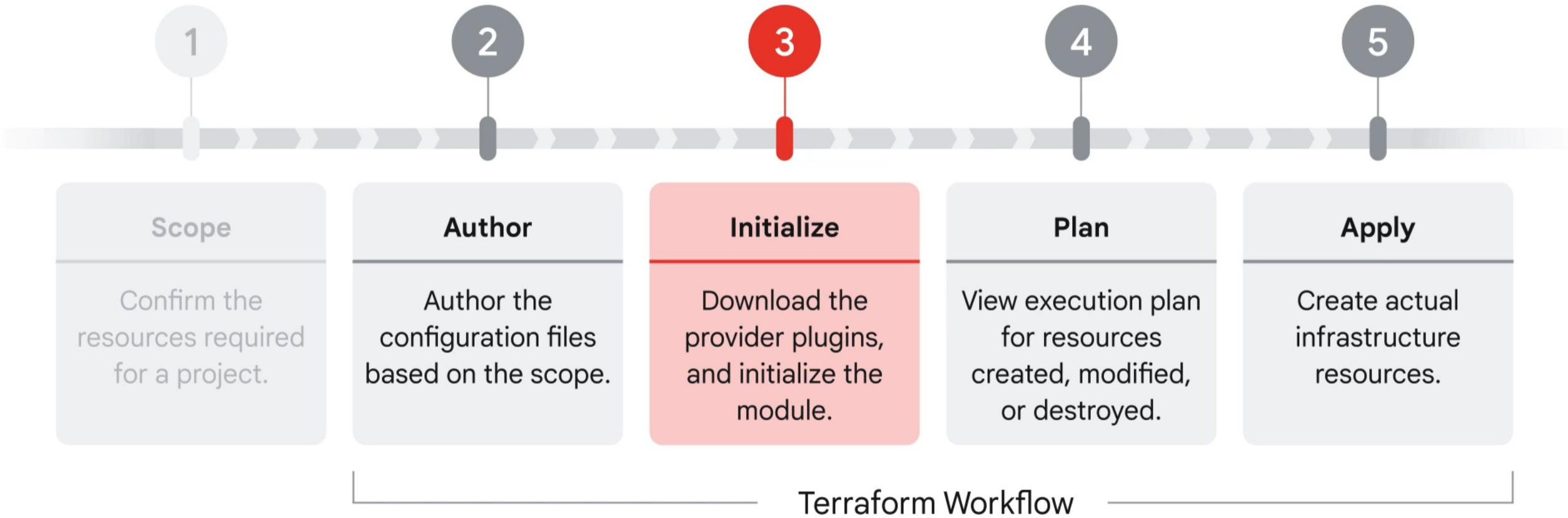
- It is the primary method for code reuse in Terraform.
- There are 2 kinds of sources:
  - Local: Source within your directory
  - Remote: Source outside your directory.



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`



Initialize Phase:

`terraform init` downloads the provider plugins

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

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

Initialize Phase:

`terraform init` downloads the provider plugins

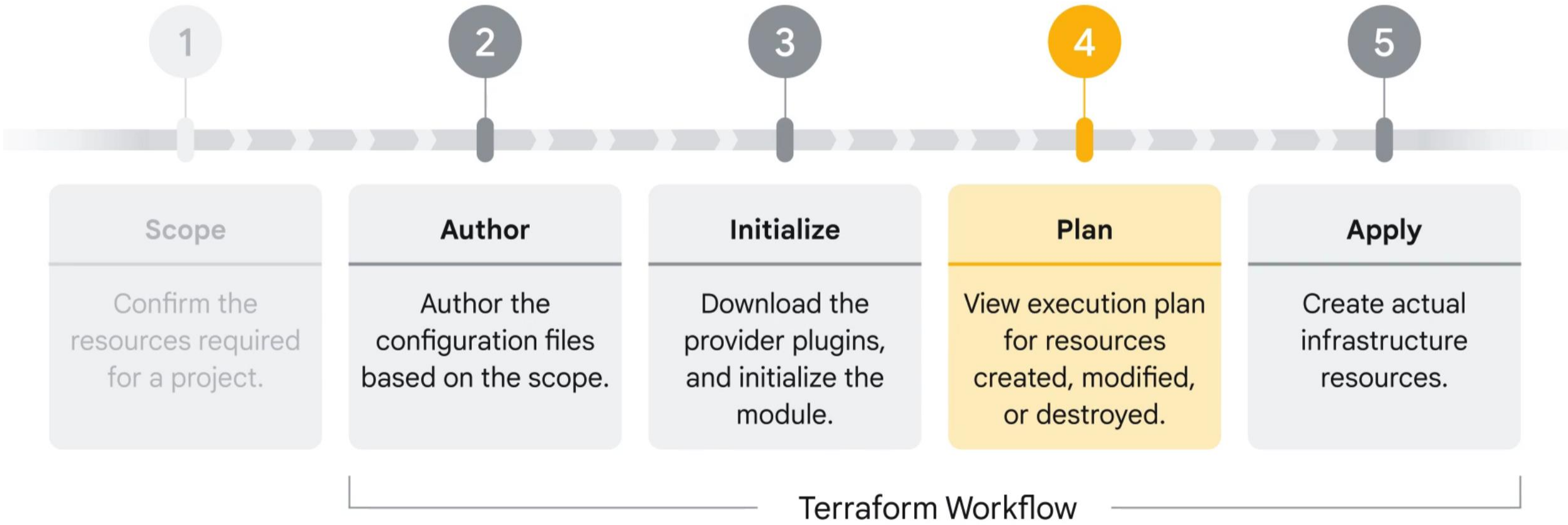
```
-- main.tf
    -- servers/
        -- main.tf
        -- variables.tf
        -- outputs.tf
```

```
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`



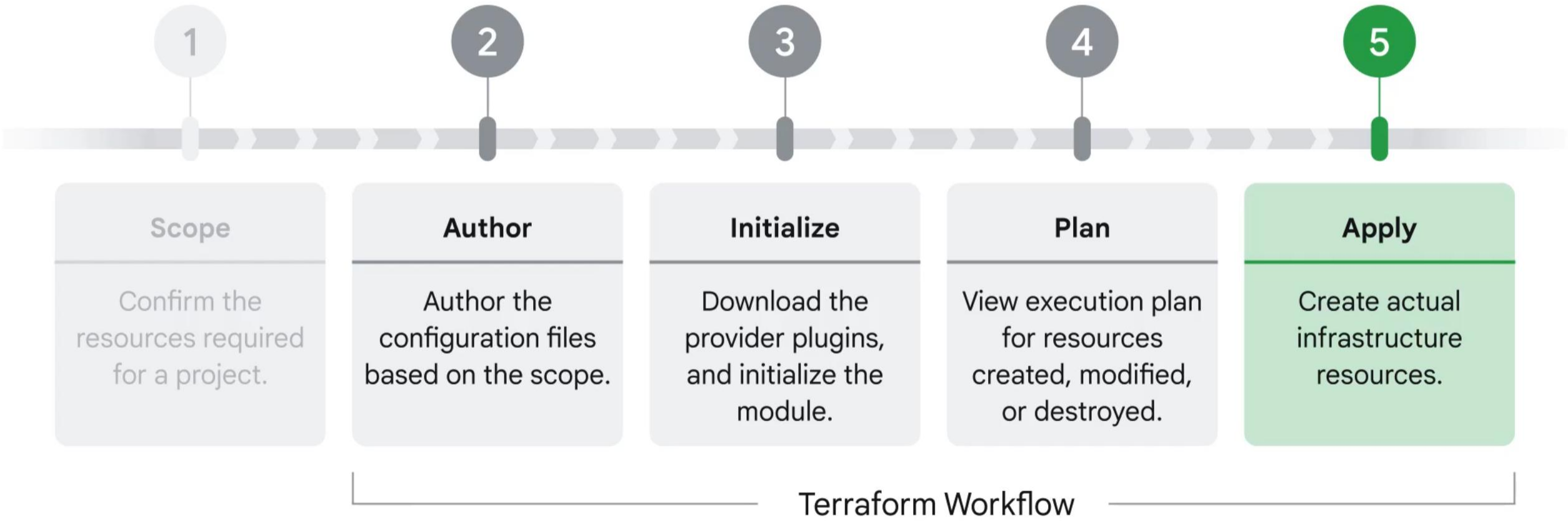
Plan Phase:

`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`



Appy 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.

```
resource “google_storge_bucket” “mybucket” {
   boot_disk { #nested arguments above
       initialize_params {
         image = "debian-cloud/debian-9"
       }
     }
   count = 2 #meta-argument in between
   machine_type=e2_micro #unaligned equal signs
   ..
}
```

```
resource “google_storge_bucket” “mybucket” {
   count        = 2 #meta-argument last

   machine_type    = e2_micro #align equal signs

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

terraform fmt

Terraform *fmt* can enforce code convention best practices.

Before `terraform fmt`:

```
resource “google_storge_bucket” “mybucket” {
    boot_disk { #nested arguments above
        initialize_params {
          image="debian-cloud/debian-9"
        }
    }
    count = 2 #meta-argument in between
    machine_type=e2_micro #unaligned equal signs
    ..
```

After `terraform fmt`:

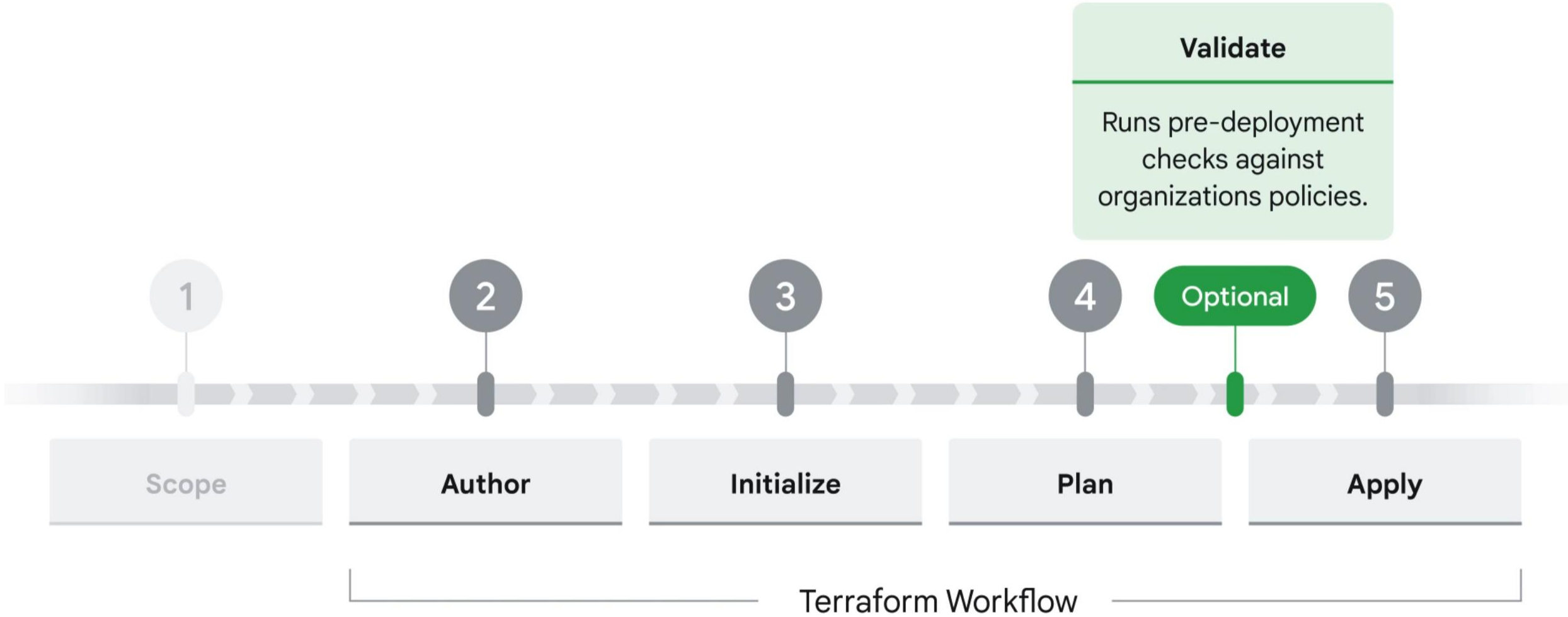
```
resource “google_storge_bucket” “mybucket” {
    count            = 2 #meta-argument first
    machine_type     = e2_micro #align equal signs
    #line space before a nested block

    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 validate phase



Why use the Terraform validator tool?

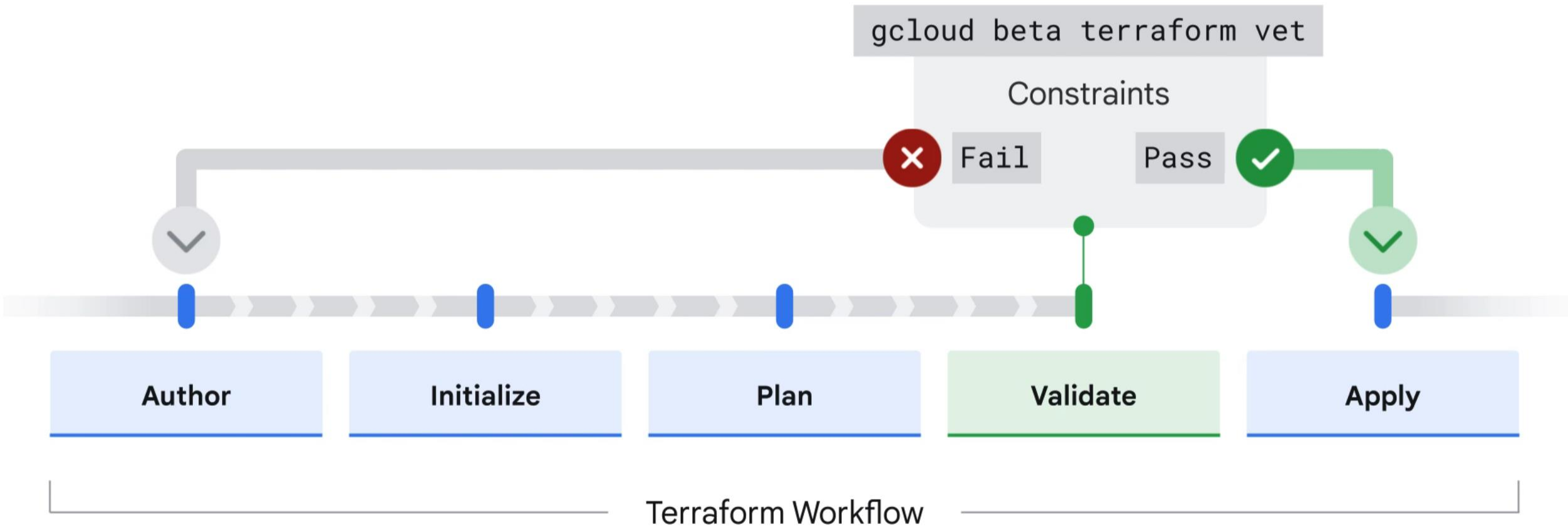
Security and governance team set **constraints**.

Automate validation

Governance policies

Security violation

The Terraform Validator



Terraform Validator Benefits

Enforce policies	Enforce policies at any stage of application development
Remove manual errors	Remove manual errors by automating policy validation
Reduce time to learn	Reduce learning time by using a single paradigm for all policy management

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.

Potential Improvements

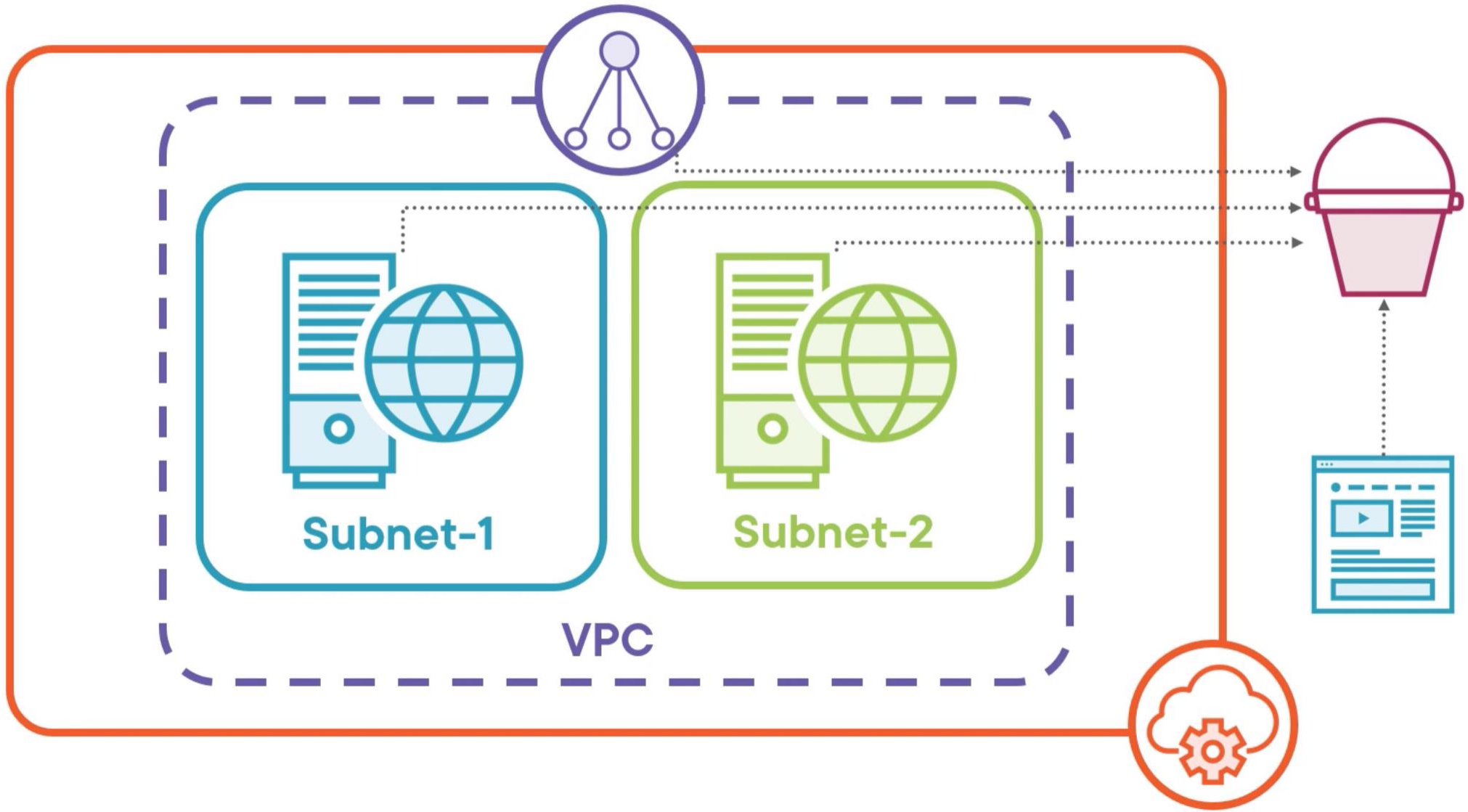
Dynamically increase instances

Use a template for startup script

Simplify networking input

Add consistent naming prefix

Deployment Architecture



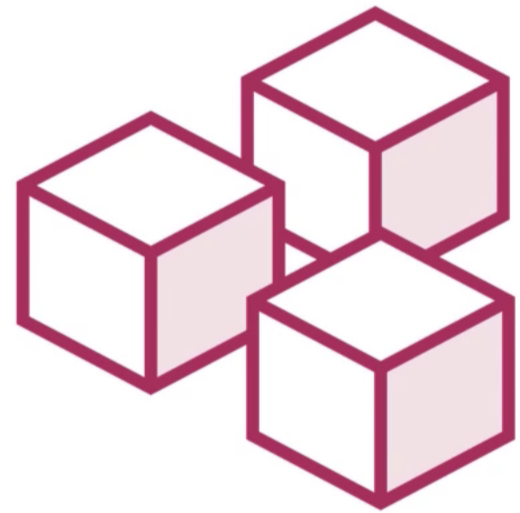
Looping Constructs

Count

Integer

For_each

Map or set



Dynamic blocks

Map or set

```
resource "aws_instance" "web_servers" {
  count = 3
  tags = {
    Name = "globo-web-${count.index}"
  }
}
```

Count References

```
<resource_type>.<name_label>[element].<attribute>
aws_instance.web_server[0].name # Single instance
aws_instance.web_server[*].name # All instances
```

For_each Syntax

s3.tf

```
resource "aws_s3_bucket_object" "taco_toppings" {
  for_each = {
    cheese = "cheese.png"
    lettuce = "lettuce.png"
  }
  key    = each.value
  source = ".${each.value}"
  tags = {
    Name = each.key
  }
```

```
resource "aws_s3_bucket_object" "taco_toppings" {
  for_each = {
    cheese = "cheese.png"
    lettuce = "lettuce.png"
  }
}
```

For_each References

```
<resource_type>.<name_label>[key].<attribute>

aws_s3_bucket_object.taco_toppings["cheese"].id # Single instance

aws_s3_bucket_object.taco_toppings[*].id # All instances
```

Terraform Functions and Expressions

Terraform Expressions

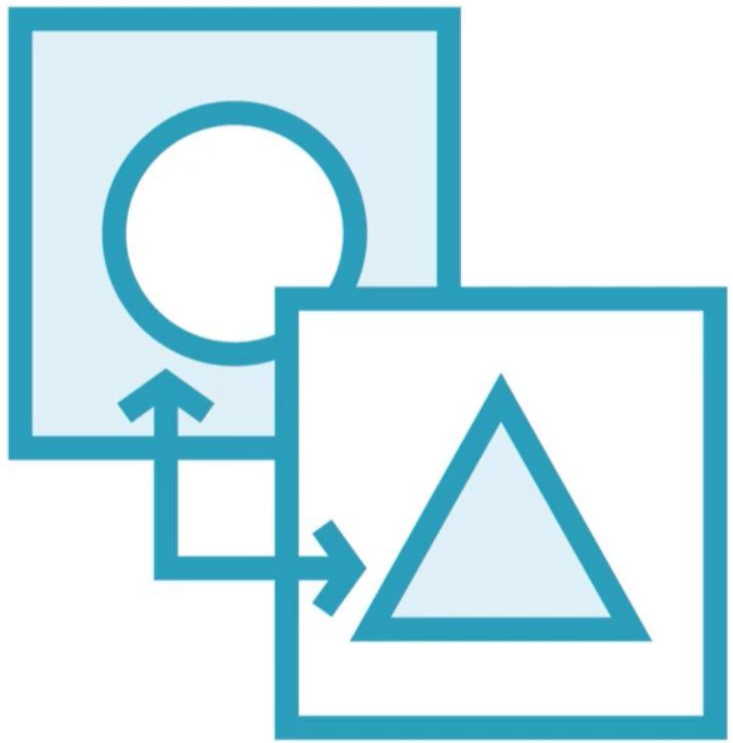
Interpolation and heredoc

Arithmetic and logical operators

Conditional expressions

For expressions

Terraform Functions



Built-in Terraform

Func_name(arg1, arg2, arg3, ...)

Test in terraform console

Several broad categories

Common Function Categories

Numeric

min(42, 13, 7)

String

lower("TACOS")

Collection

merge(map1, map2)

IP network

cidrsubnet()

Filesystem

file(path)

Type Conversion

toset()

Functions to Use

```
# Startup script
templatefile(file_location, { map of variables })
# Extract subnet address from VPC CIDR
cidrsubnet(cidr_range, subnet bits to add, network number)
# Add tags to common tags
merge(common_tags, { map of additional tags })
# S3 bucket name
lower("bucket name")
```

More Teams, More Problems

Information Security

Define roles, policies, and groups

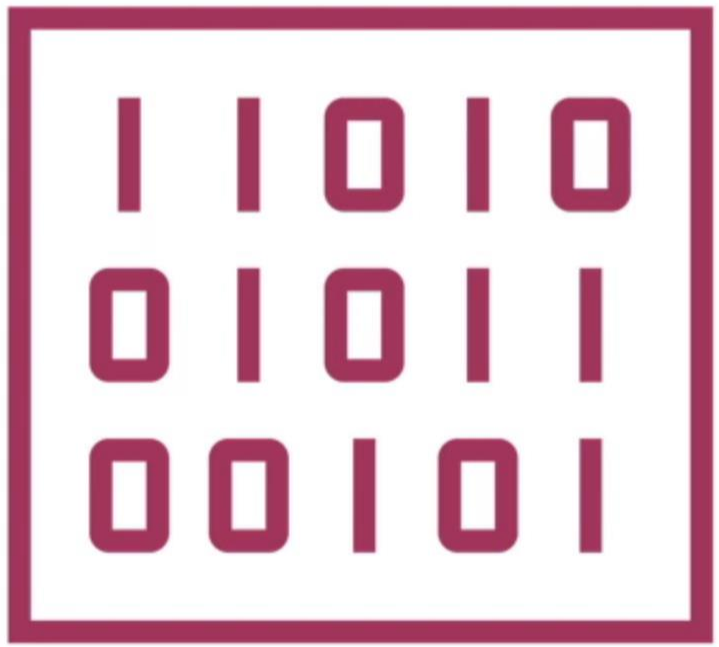
Software Development

Read network configuration for app deployment

Change Management

Store configuration data centrally

Data Sources



Glue for multiple configurations

Resources are data sources

Providers have data sources

Alternate data sources

- Templates
- HTTP
- External
- Consul

HTTP Data Source

```
# Example data source
data "http" "my_ip" {
  url = "http://ifconfig.me"
}
# Using the response
data.http.my_ip.body
```

Templates

Manipulation of strings

Overloaded term

- Quoted strings
- Heredoc syntax
- Provider
- Function

Interpolation and directives

```
# Simple interpolation
"${var.prefix}-app"

# Conditional directive
"%{ if var.prefix != ""}${var.prefix}-app%{ else }generic-app%{ endif }"

# Collection directive with heredoc
<<EOT
%{ for name in local.names }
${name}-app
%{ endfor }
EOT
```

Template strings

You've been using these implicitly

Expressed in the configuration directly

Heredoc preferred for readability

Template Syntax In-line

```
# Template data source
data "template_file" "example" {

  count = "2"

  template = "$${var1}-$${current_count}"

  vars = {

    var1 = var.some_string
    current_count = count.index
  }
}

# Using the template
data.template_file.example.rendered
```

```
# Template configuration
data "template_file" "peer-role" {
  template = file("peer_policy.txt")
  vars = {
    vpc_arn = var.vpc_arn
  }
}

# Or templatefile function
templatefile("peer_policy.txt",
    { vpc_arn = var.vpc_arn } )
```

peer_policy.txt

```
{
  "Version": "2012-10-17",
  "Statement": [
   {
    "Action": [
     "ec2:AcceptVpcPeeringConnection",
     "ec2:DescribeVpcPeeringConnections"
    ],
    "Effect": "Allow",
    "Resource": [
     "${vpc_arn}"
    ]
   }
  ]
 }
```

Applying Policy as Code

HashiCorp Sentinel

Policy as code

Version control

Native policy language

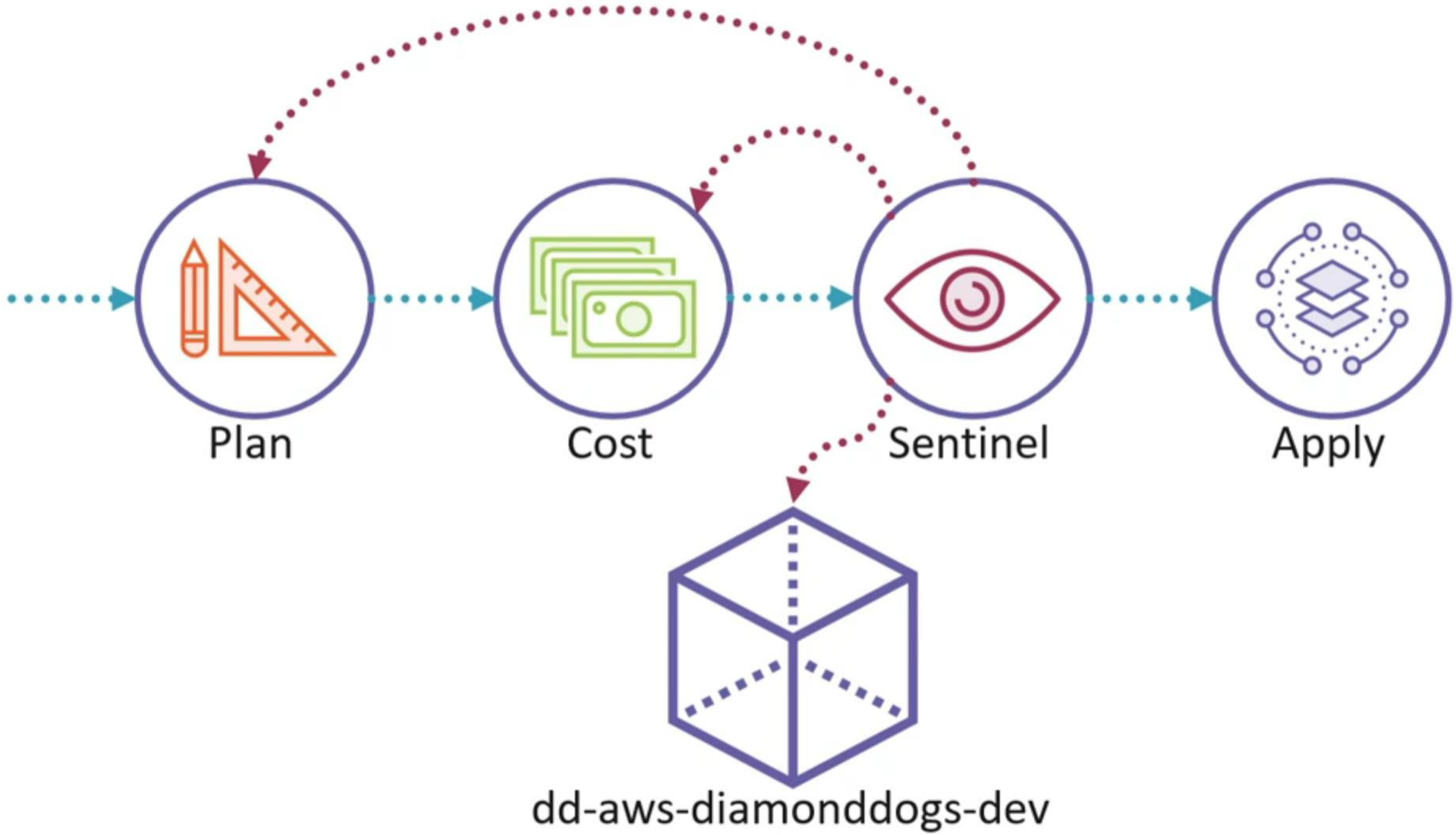
Fine grained and conditional

Import data sources

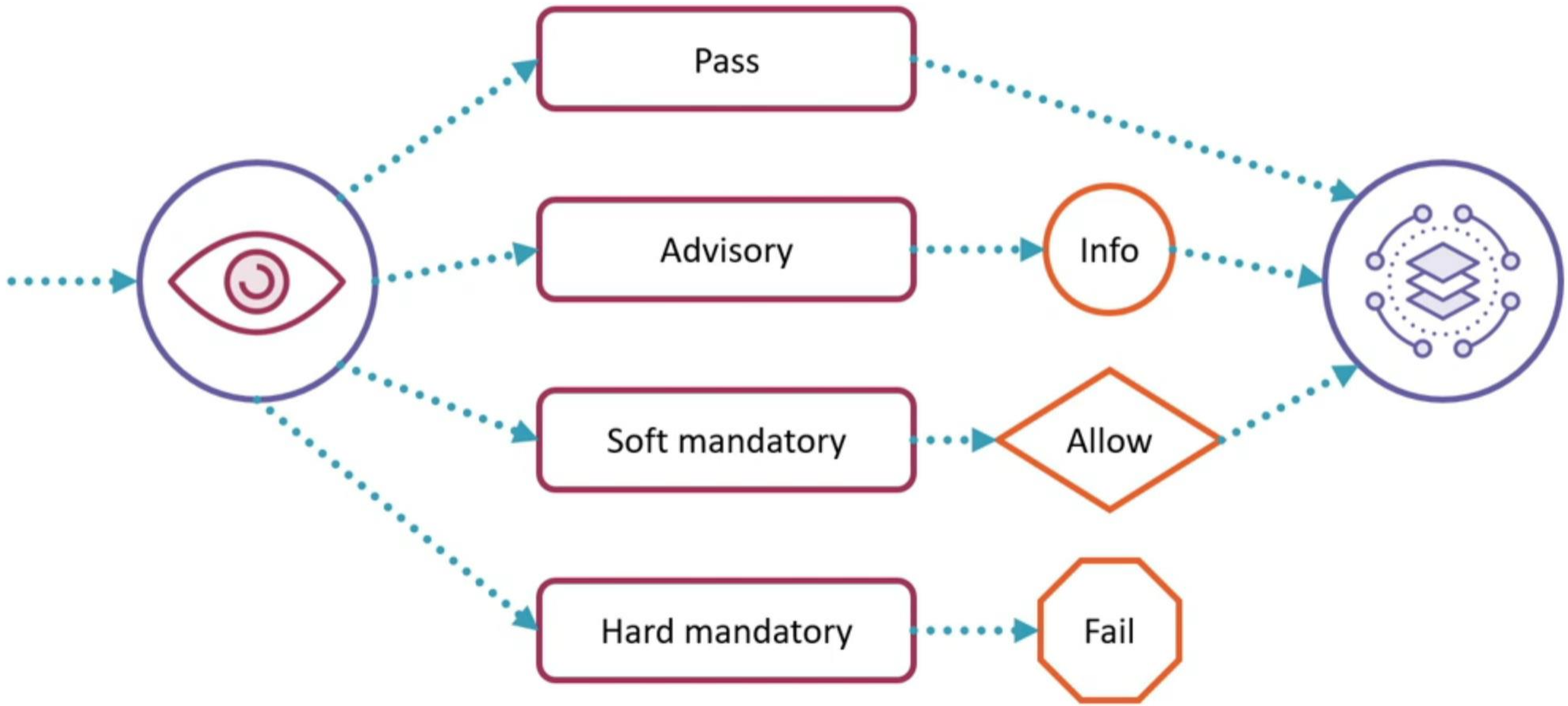
Enforcement levels

Sentinel on Terraform Cloud

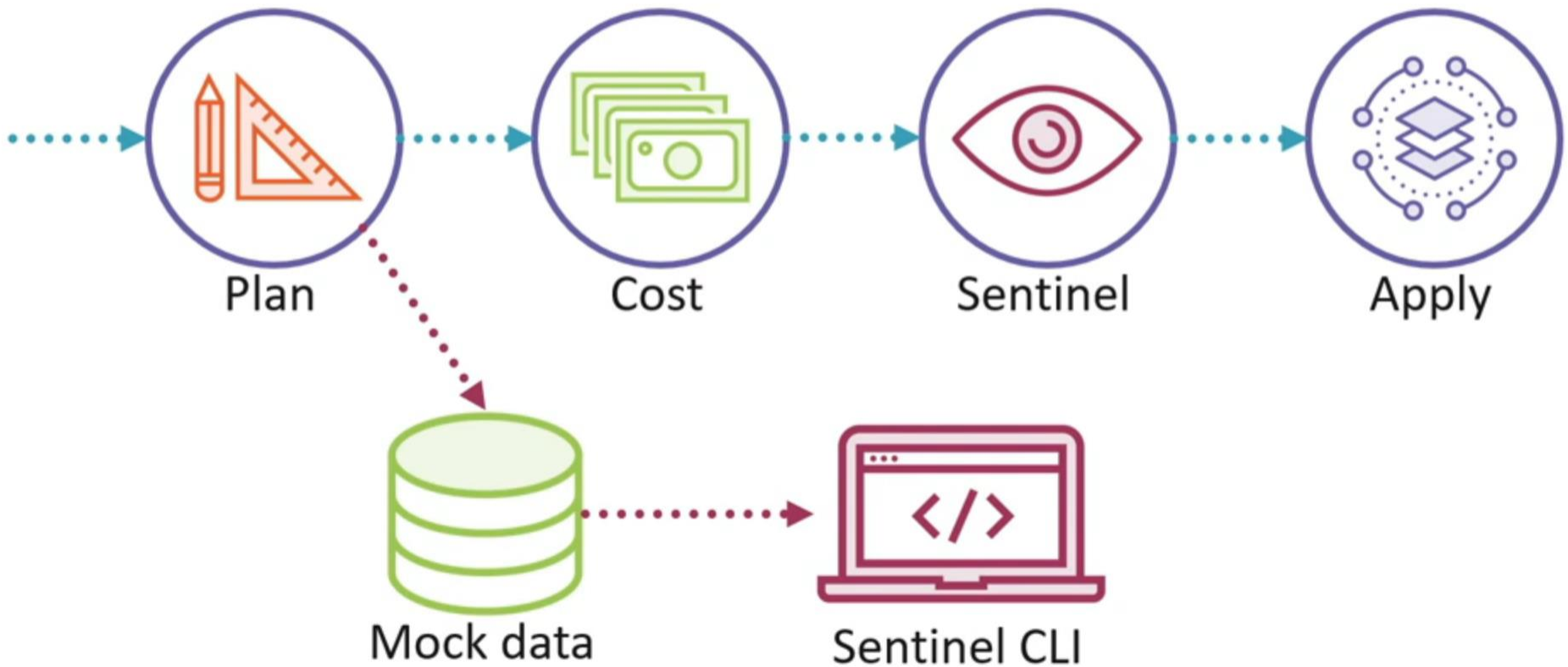
Policy Evaluation



Enforcement Actions



Testing Sentinel



instance-size.sentinel

```
# Import tfplan
import "tfplan/v2" as tfplan

# Get all EC2 instances
ec2_instances = filter tfplan.resource_changes as _, rc {
  rc.type is "aws_instance" and rc.mode is "managed" }

# Test the instance size
main = rule { all ec2_instances as _, instance {
  instance.change.after.instance_type in ["t2.micro","t2.small"] }
}
```

sentinel.hcl

```
policy "instance-type" {
  enforcement_level = "hard-mandatory"
}

policy "instance-tags" {
  source = "https://raw.githubusercontent.com/.../instance-tags.sentinel"
  enforcement_level = "advisory"
}
```

Using Modules

Leverage existing function libraries

sentinel.hcl

```
module "tfplan-functions" {

  source =
"https://raw.githubusercontent/.../
  tfplan-functions.sentinel"

}
```

instance-tags.sentinel

```
import "tfplan-functions" as plan


ec2_instances =
  plan.find_resources("aws_instance")
```

Questions ???