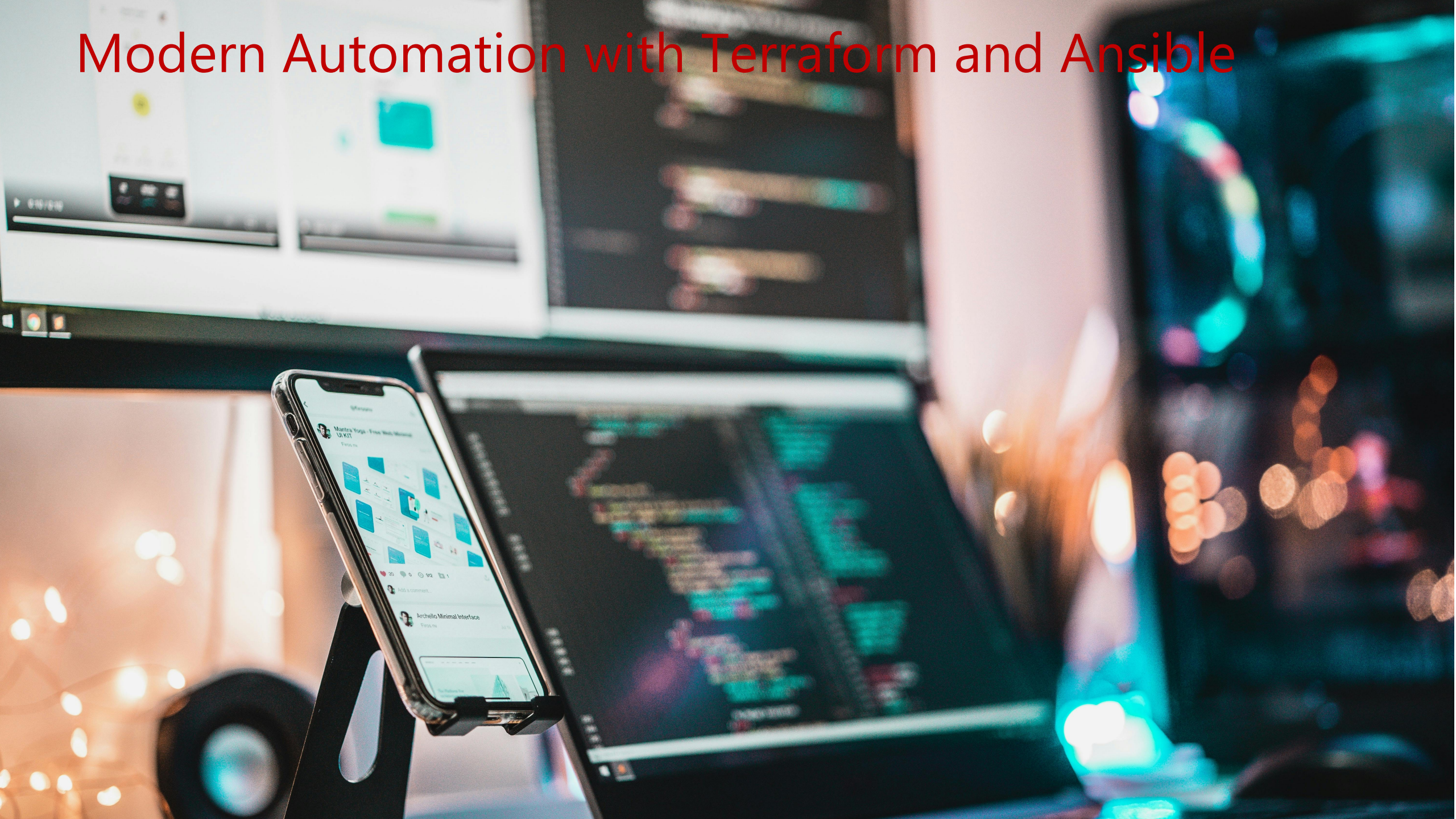
Modern Automation with Terraform and Ansible



TEKsystems Global Services

WORKFORCE DEVELOPMENT

Lab page

https://jruels.github.io/modern-automation/

Terraform configuration

```
##Amazon Infrastructure
provider "aws" {
  region = "${var.aws_region}"
}
```

- Provider defines what infrastructure Terraform will be managing
  - Pass variables to provider
    - Region, Flavor, Credentials

Terraform configuration

```
data "aws_ami" "ubuntu" {
  most_recent = true
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }
  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
  owners = ["099720109477"] # Canonical
}
```

- Data sources
  - Queries AWS API for latest Ubuntu 16.04 image.
  - Stores results in a variable which is used later.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Resource: Defines specifications for creation of infrastructure.

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- ami is set by results of previous data source query

Terraform configuration

```
##AWS Specific Vars
variable "aws_master_count" {
  default = 10
}
variable "aws_worker_count" {
  default = 20
}
variable "aws_key_name" {
  default = "k8s"
}
variable "aws_instance_size" {
  default = "t2.small"
}
variable "aws_region" {
  default = "us-west-1"
}
```

- Define sane defaults in variables.tf

Terraform configuration

```
resource "aws_instance" "aws-k8s-master" {
  subnet_id              = "${var.aws_subnet_id}"
  depends_on             = ["aws_security_group.k8s_sg"]
  ami                    = "${data.aws_ami.ubuntu.id}"
  instance_type          = "${var.aws_instance_size}"
  vpc_security_group_ids = ["${aws_security_group.k8s_sg.id}"]
  key_name               = "${var.aws_key_name}"
  count                  = "${var.aws_master_count}"
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 20
    delete_on_termination = true
  }
  tags {
    Name = "k8s-master-${count.index}"
    role = "k8s-master"
  }
}
```

- Value is 'count' is setup by variable in variables.tf

Terraform CLI

The `terraform plan -refresh-only` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure but does modify the state file. If the state is changed, this may cause changes to occur during the next `plan` or `apply`.

Terraform CLI

Terraform is normally run from inside the directory containing the *.tf files for the root module. Terraform checks that directory and automatically executes them.

In some cases, it makes sense to run the Terraform commands from a different directory. This is true when wrapping Terraform with automation. To support that Terraform can use the global option `-chdir=...` which can be included before the name of the subcommand.

Terraform CLI

The `chdir` option instructs Terraform to change its working directory to the given directory before running the given subcommand. This means that any files that Terraform would normally read or write in the current working directory will be read or written in the given directory instead.

Terraform CLI

Command:

```
terraform -chdir=environments/dev (apply|plan|destroy)
```

Output:

```
❯ terraform apply
aws_security_group.k8s_sg: Creating...
  arn:                                      "" => "<computed>"
  description:                              "" => "Allow all
inbound traffic necessary for k8s"
  egress.#:                                 "" => "1"
  egress.482069346.cidr_blocks.#:           "" => "1"
  egress.482069346.cidr_blocks.0:           "" => "0.0.0.0/0"
```

Performs the subcommand in the specified directory.

Terraform CLI

It can be time consuming to update a configuration file and run `terraform apply` repeatedly to troubleshoot expressions not working.

Terraform has a console subcommand that provides an interactive console for evaluating these expressions.

Terraform CLI

```
variable "x" {

default = [
        {
          name = "first",
          condition = {
            age = "1"
          }
          action = {
            type = "Delete"
          }
        },{
          name = "second",
          condition = {
            age = "2"
          }
          action = {
            type = "Delete"
          }
        }
    ]
}
```

Terraform CLI

Command:

```
terraform console
```

Output:

```
> var.x[1].name
"second"

var.x[0].condition
{
  "age" = "1"
}
```

Test expressions and interpolations interactively.

Terraform CLI

Terraform includes a graph command for generating visual representation of the configuration or execution plan. The output is in DOT format, which can be used by GraphViz to generate charts.

Terraform CLI

Command:

```
terraform graph
```

Output:

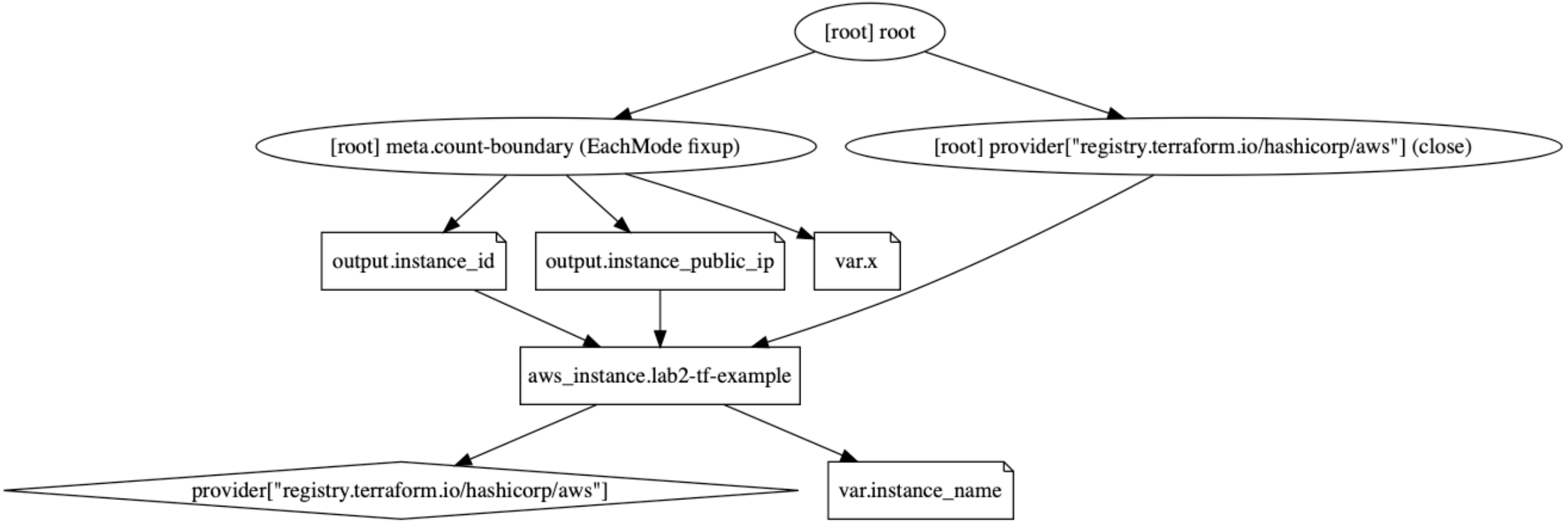
```
digraph {
        compound = "true"
        newrank = "true"
        subgraph "root" {
                "[root] aws_instance.lab2-tf-example (expand)"
[label = "aws_instance.lab2-tf-example", shape = "box"]
...
```

Create a visual graph of Terraform resources

```
terraform graph | dot -Tsvg > graph.svg
```

Terraform CLI

```
terraform graph| dot -Tsvg > graph.svg
```



HashiCorp Configuration Language (HCL)

ARM JSON:

```
"name" : "[concat(parameters('PilotServerName'), '3')]",
```

Terraform:

```
name = "${var.PilotServerName}3"
```

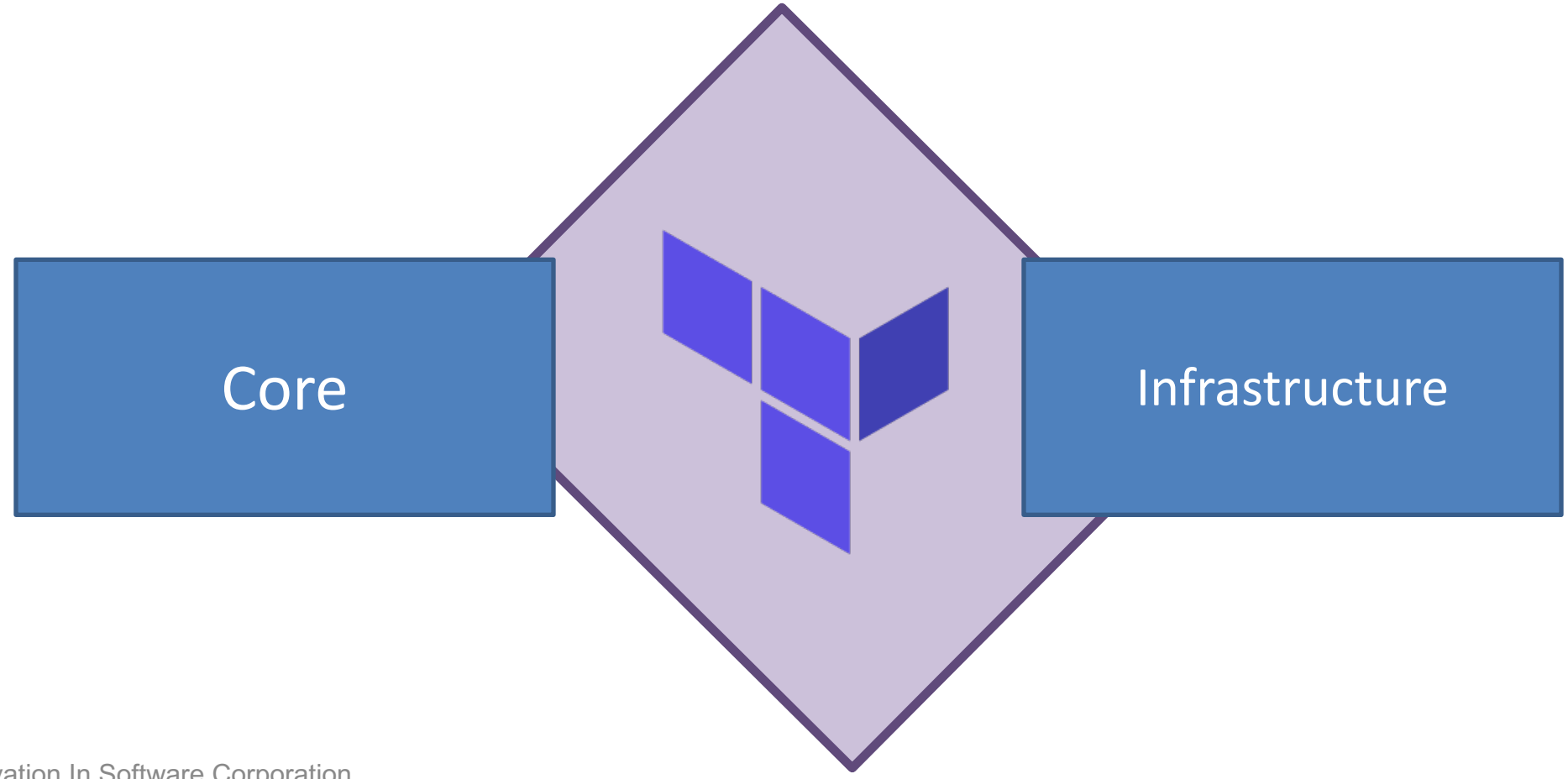
Terraform code (HCL) is "easy" to learn and "easy" to read. It is also more compact than equivalent JSON configuration.

Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers

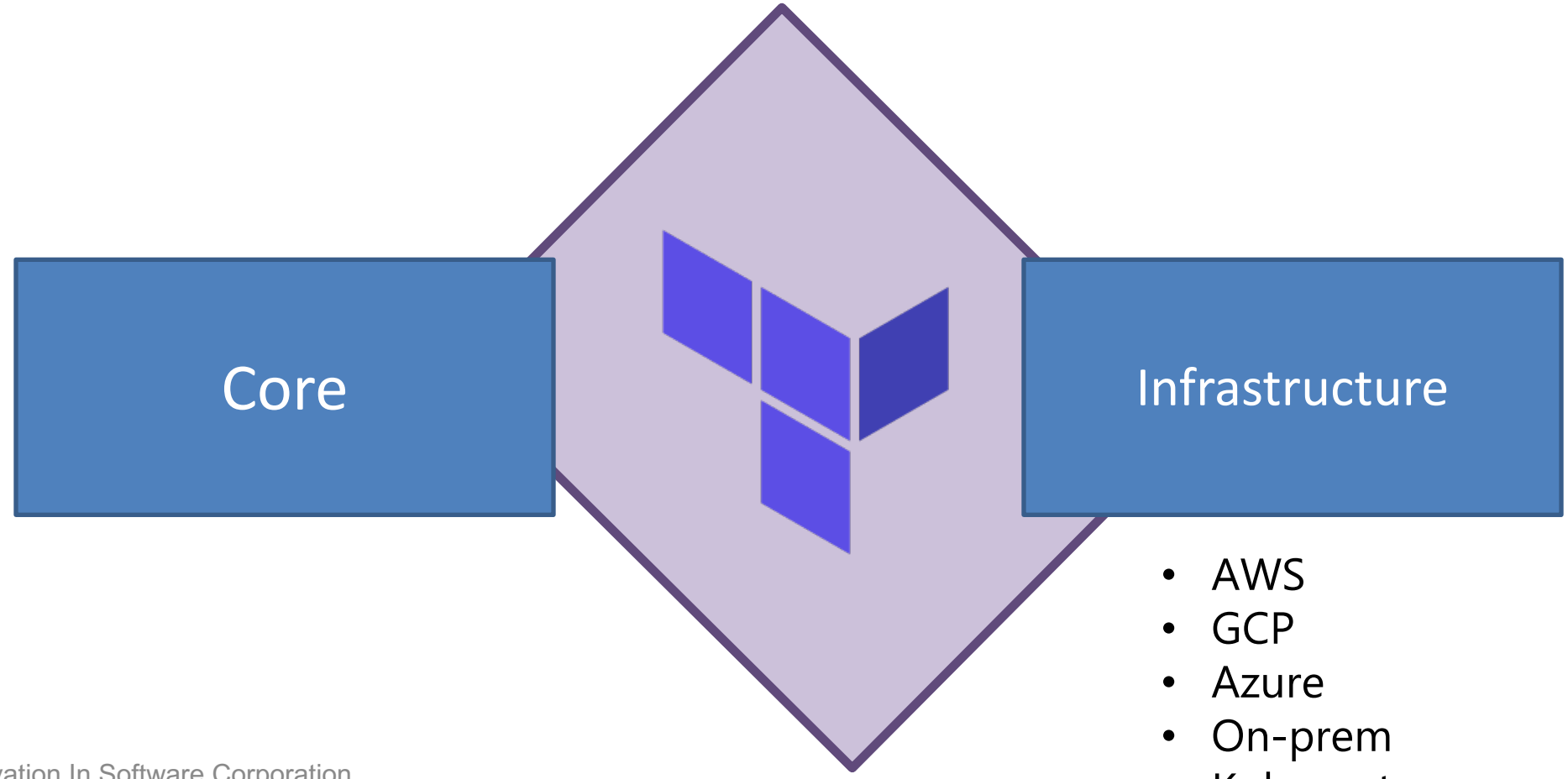
Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



Terraform model

Terraform code is broken into three parts: Core Terraform, Modules and Providers



- AWS
- GCP
- Azure
- On-prem
- Kubernetes

Terraform Resources

Resource Types Overview

```
# Cloud provider resource
resource "aws_instance" "web" {}


# Meta-resource
resource "terraform_remote_state" "network" {}


# Local-only resource
resource "null_resource" "example" {}
```

Resources in Terraform represent different kinds of infrastructure components, each with their own specific configuration patterns.

- Provider-specific resources (aws_instance)
- Meta-resources (terraform_remote_state)
- Null resources for local operations
- Data resources for read-only operations

Terraform resources

Every Terraform resource is structured the exact same way.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
```

resource = top level keyword

Resource Arguments

```
resource "aws_instance" "web" {
  # Required arguments
  ami = "ami-0c02fb55956c7d316" # Amazon Linux 2 AMI
in us-east-1
  instance_type = "t2.micro"

  # Meta-arguments
  count = 3

  tags = {
    Name = "web-server"
  }
}
```

Resources accept different types of arguments that define their configuration and behavior.

- Required arguments (name, type)
- Optional arguments with defaults
- Computed arguments (set by provider)
- Meta-arguments (count, for_each)

Lab: Build first instance

Terraform local-only resources

While most resource types correspond to an infrastructure object type that is managed via a remote network API, there are certain specialized resource types that operate only within Terraform itself, calculating some results and saving those results in the state for future use.

For example, you can use local-only resources for:

- Generating private keys
- Issue self-signed TLS certs
- Generating random ids

- The behavior of local-only resources is the same as all other resources, but their result data exists only within the Terraform state. "Destroying" such a resource means only to remove it from the state, discarding its data.

Terraform local-only resources

Local-only resources are also referred to as logical resources

This is primarily used for easy bootstrapping of throwaway development environments.

```
resource "tls_private_key" "example" {
  algorithm   = "ECDSA"
  ecdsa_curve = "P384"
}
```

Local-Only Resources - Random

```
resource "random_id" "bucket_suffix" {
  byte_length = 8
}


resource "random_password" "db_password" {
  length = 16
  special = true
  override_special = "!#$%&*()-_=+[]{}<>:?"
}
```

Random resources generate and maintain consistent random values
across Terraform runs.

- Random strings
- Random integers
- UUIDs
- Passwords

Local-Only Resources - Time

```
resource "time_rotating" "key_rotation" {
  rotation_days = 30
}


resource "aws_kms_key" "example" {
  description = "crypto-key"
  rotation_enabled = true
}
```

Time resources help manage time-based operations and dependencies.

- Rotation triggers
- Delayed operations
- Time-based dependencies
- Schedule tracking

Terraform variables

Input variables serve as parameters for a Terraform module, allowing aspects of the module to be customized without altering the module's own source code, and allowing modules to be shared between different configurations.

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.

Terraform variables

Each input variable accepted by a module must be declared using a variable block.

The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform variables

Terraform has some reserved variables:

- source
- version
- providers
- count
- for_each
- lifecycle
- depends_on
- locals

- These are reserved for meta-arguments in module blocks.

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Terraform variables

Optional variable arguments:

- default: A default value which makes the variable optional
- type: Specifies value type accepted for the variable.
- description: Specifies the variable's documentation
- validation: A block to define validation rules
- sensitive: Does not show value in output

Environment-Specific Variable Files

```
# dev.tfvars
environment = "dev"
project_id = "my-project-dev"
node_count = 1
instance_type = "t3.medium"


# prod.tfvars
environment = "prod"
project_id = "my-project-prod"
node_count = 3
instance_type = "t3.large"
```

Using separate tfvars files allows managing multiple environments with the same code base.

- One configuration, multiple environments
- Clear separation of config from code
- Easy environment promotion
- Consistent resource naming

Local Value Patterns

```
locals {
  resource_prefix = "${var.environment}-
${var.project_id}"
  common_tags = {
    Environment = var.environment
    Project = var.project_id
    ManagedBy = "terraform"
  }
  instance_name = "${local.resource_prefix}-
instance"
  bucket_name = "${local.resource_prefix}-
storage"
}
```

Local values help create consistent naming and tagging conventions across resources.

- Combine multiple variables
- Create reusable patterns
- Enforce naming conventions
- Reduce repetition

Output Management

```
output "environment_info" {
  value = {
    name = var.environment


    resources = {
      instances = aws_instance.web[*].tags["Name"]
      bucket = aws_s3_bucket.data.bucket
      vpc = aws_vpc.main.tags["Name"]
  }

  endpoints = {
    api = "https://${local.resource_prefix}-
api.example.com"
    web = "https://${local.resource_prefix}-
web.example.com"
    }
  }
}
```

Outputs expose environment-specific values while maintaining consistency across environments.

- Environment-specific values
- Consistent output structure
- Cross-environment references
- Pipeline integration points

Environment-Specific Conditions

```
locals {
  is_production = var.environment == "prod"
  instance_count = {
    dev = 1
    staging = 2
    prod = 3
  }
}
```

Use variables to create environment-specific resource configurations.

- Conditional resource creation
- Environment-based sizing
- Feature flags
- Resource counts

Terraform provisioners

Terraform is designed to programmatically create and manage infrastructure. It is not intended to replace Ansible, Chef or any other configuration management tool.

It does include provisioners as a way to prepare the system for your services.

Generic provisioners:

- file
- local-exec
- remote-exec

Provisioner Overview

```
resource "google_compute_instance" "web" {
  name = "web-server"
  machine_type = "e2-micro"


provisioner "remote-exec" {
  inline = [
    "apt-get update",
    "apt-get install -y nginx"
    ]
  }
}
```

Provisioners allow you to execute actions on local or remote machines as part of resource creation or destruction.

- Post-creation configuration
- Software installation
- File transfers
- Command execution

Terraform provisioners

Generic Provisioners:

- file
  - The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource. The file provisioner supports both SSH and WinRM type connections.
- local-exec
  - The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.
- remote-exec
  - The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. The remote-exec provisioner supports both SSH and WinRM type connections.

Remote-Exec Provisioner

```
resource "google_compute_instance" "web" {
  name = "web-server"

  connection {
    type = "ssh"
    user = "admin"
    private_key =
file("${path.module}/ssh/private_key")
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

Remote-exec provisioners run commands on the remote resource after creation.

- Install software packages
- Configure services
- Start applications
- Run setup scripts

Local-Exec Provisioner

```
resource "google_compute_instance" "db" {
  name = "database"


  provisioner "local-exec" {
    command = <<-EOT
    echo
"DB_HOST=${self.network_interface[0].network_ip}" >>
.env
    ./scripts/update-dns.sh ${self.name}
${self.network_interface[0].access_config[0].nat_ip}
    EOT
  }
}
```

Local-exec provisioners run commands on the machine executing Terraform.

- Generate configuration files
- Run local scripts
- Trigger external tools
- Update documentation

File Provisioner

```
resource "google_compute_instance" "app" {
  name = "application"


  connection {
    type = "ssh"
    user = "admin"
    private_key = file(var.ssh_private_key)
  }

  provisioner "file" {
    source = "configs/"
    destination = "/etc/application/"
  }
}
```

File provisioners copy files or directories to the remote resource.

- Configuration files
- Scripts
- Application code
- SSL certificates

Lab: Working with variables

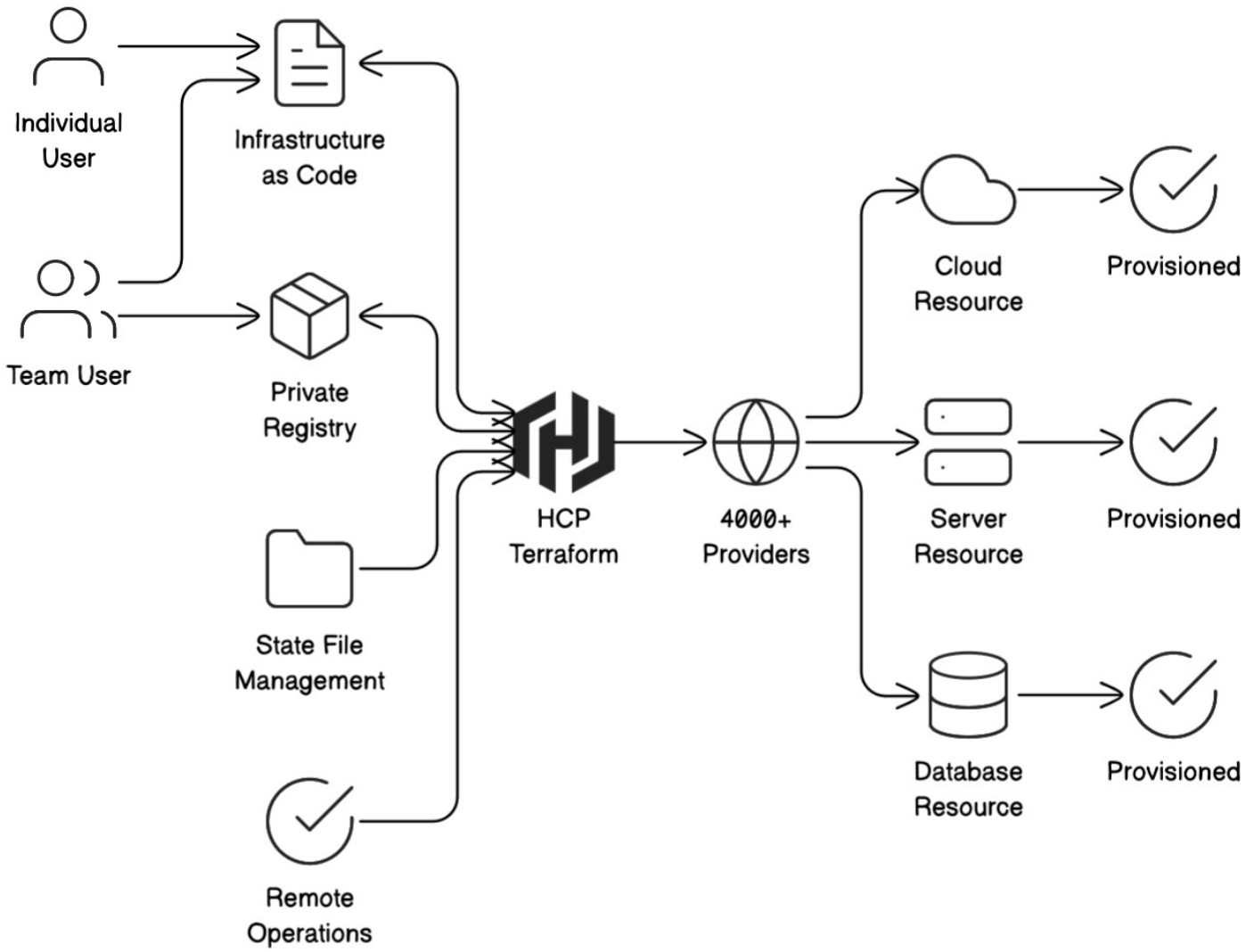
Lab: Multi resource deployment

What is HCP Terraform?

HCP Terraform is a managed service that brings the power of Terraform's infrastructure as code workflow to the cloud. Instead of running Terraform locally, HCP Terraform provides a secure, consistent, and collaborative environment for managing infrastructure. It centralizes state management, secrets, and policy enforcement, and integrates with version control systems to streamline infrastructure development.

By using HCP Terraform, teams can automate, govern, and audit their infrastructure changes with greater confidence and efficiency.

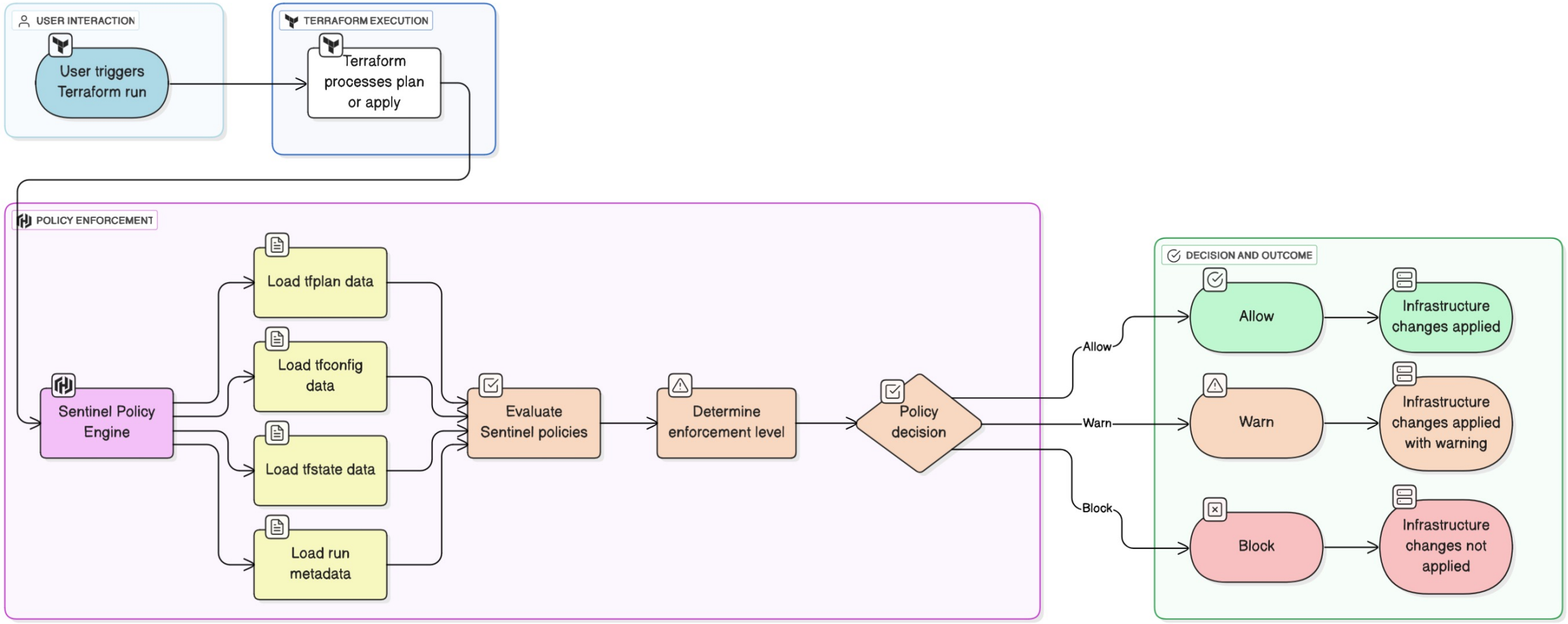
HCP Terraform Overview



Key Features of HCP Terraform

- Centralized and secure storage of Terraform state files and secrets
- Integration with version control systems (VCS) for automated runs
- Private registry for sharing modules and providers within your organization
- Enhanced visibility into infrastructure changes and operations
- Access controls, policy enforcement, and cost estimation (paid features)
- Collaboration tools for team-based infrastructure management

HCP Terraform Workflow



HCP Terraform Workflows

CLI-driven workflow:

- Run Terraform commands from your terminal, with operations executed remotely in HCP Terraform's secure environment.

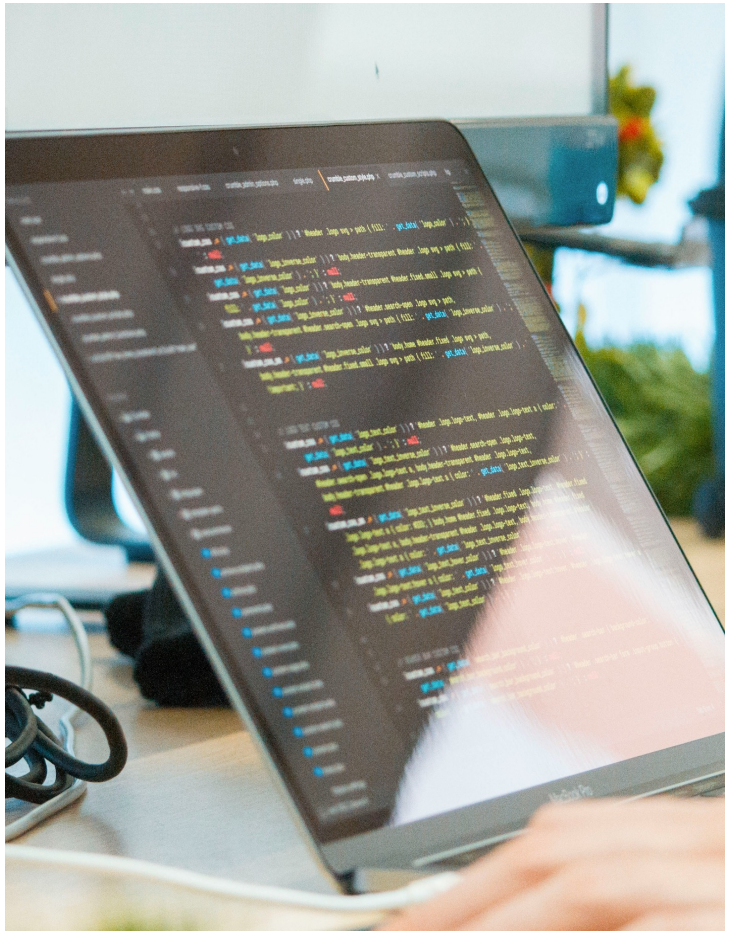
VCS-driven workflow:

- Connect a GitHub, GitLab, or Bitbucket repository; changes to code trigger runs automatically.

API-driven workflow:

- Programmatically interact with HCP Terraform using its API for custom automation and integrations.

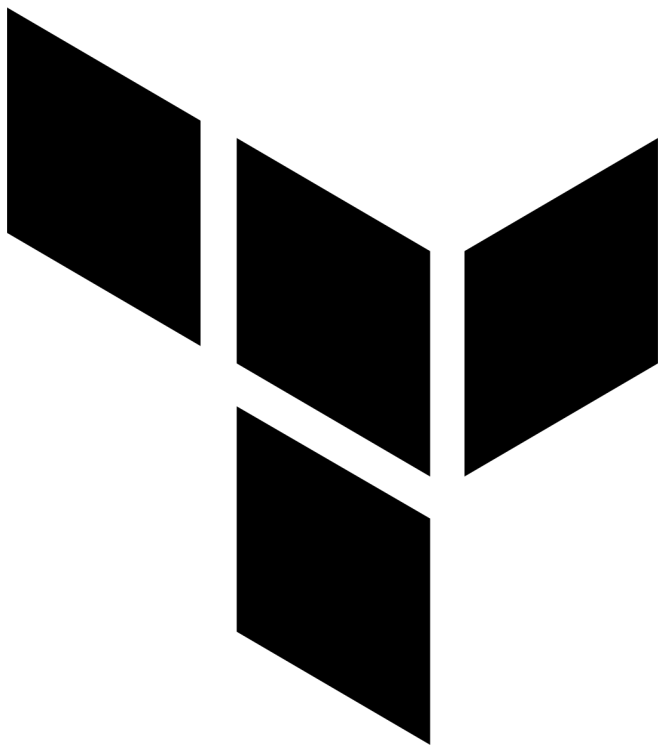
What is Terraform Enterprise?



Terraform Enterprise is HashiCorp's self-hosted distribution of HCP Terraform, designed for organizations that require full control over their infrastructure automation platform. By running Terraform Enterprise on your own infrastructure, you gain access to all the features of HCP Terraform, including advanced policy enforcement, private module registries, and detailed audit logging, while maintaining data residency and compliance with internal security requirements.

Terraform Enterprise is ideal for enterprises with strict regulatory, security, or operational needs that cannot be met by a fully managed cloud service.

Key Features of Terraform Enterprise



- Self-hosted, private instance of the HCP Terraform application
- No resource limits—scale to meet enterprise needs
- Enterprise-grade features:
- Audit logging for compliance and traceability
- SAML single sign-on (SSO) for secure user authentication
- Private module and provider registries
- Advanced access controls and team management
- Supports integration with existing enterprise systems and workflows
- Provides the same core functionality as HCP Terraform, with additional control and flexibility for on-premises or private cloud deployments

Focus of This Course – HCP Terraform

While Terraform Enterprise and HCP Terraform share many features and capabilities, this course will focus specifically on HCP Terraform, HashiCorp's managed cloud offering. HCP Terraform provides a streamlined, fully managed environment for running Terraform workflows, collaborating with teams, and enforcing policy as code—all without the operational overhead of managing your own infrastructure. Throughout this course, you'll learn how to leverage HCP Terraform's cloud-native features for secure, scalable, and collaborative infrastructure automation.

HCP Terraform CLI Workflow

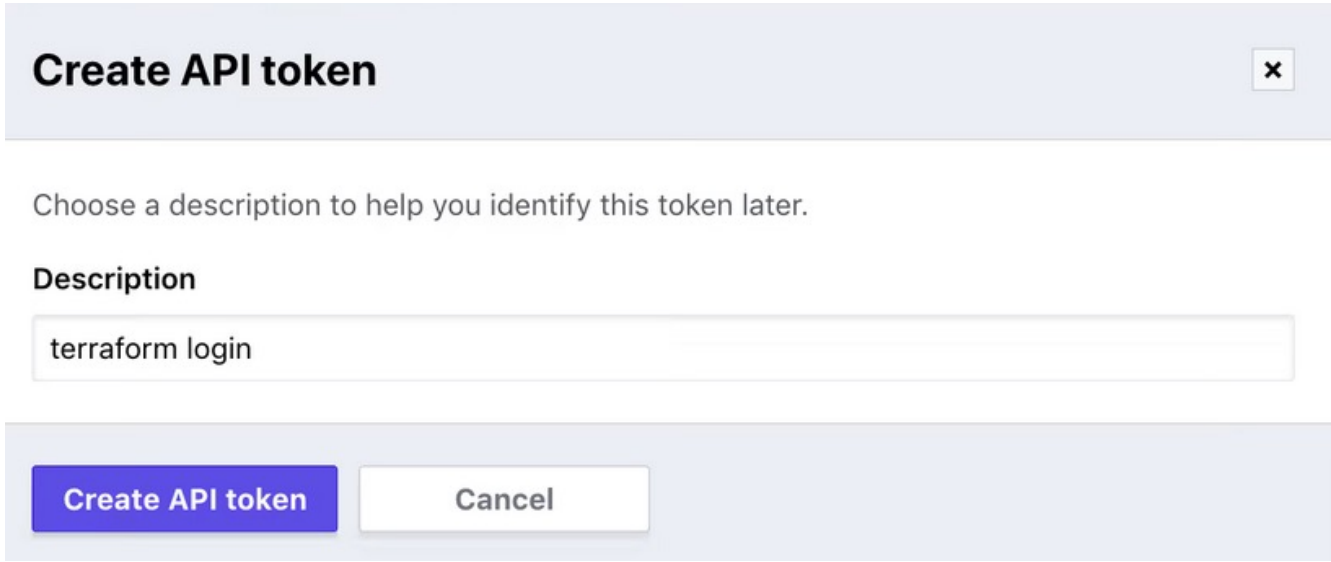
- HCP Terraform supports CLI-driven, VCS-driven, and API-driven workflows.
- The CLI workflow lets you run Terraform commands locally while using HCP Terraform for secure remote execution and state management.
- To begin, authenticate your CLI by running terraform login.
- This command links your local environment to your HCP Terraform account, enabling remote plans, applies, and collaboration from the terminal.

HCP Terraform CLI Workflow

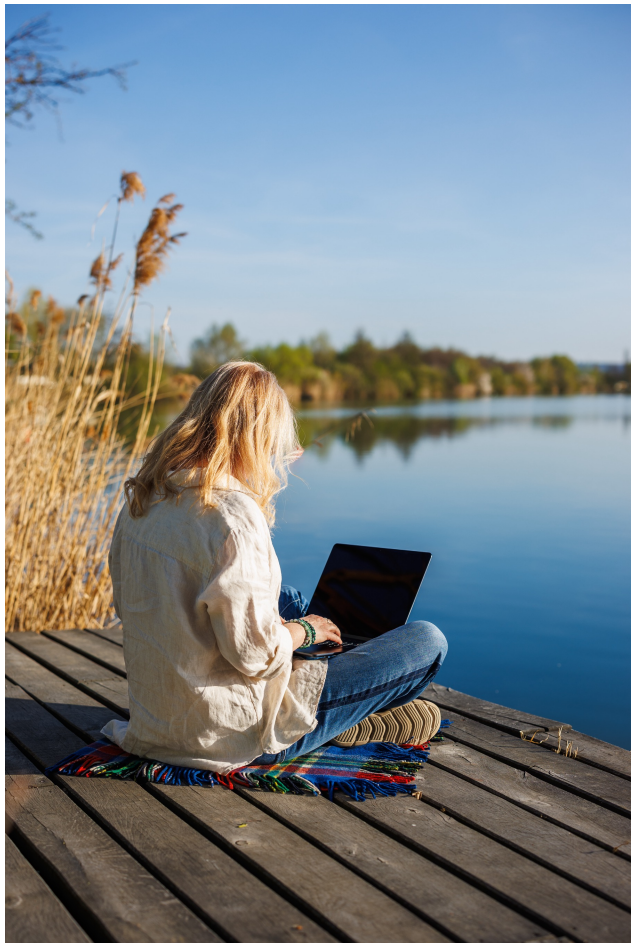
```
$ terraform login
Terraform will request an API token for app.terraform.io using your browser.

If login is successful, Terraform will store the token in plain text in
the following file for use by subsequent commands:
    /Users/redacted/.terraform.d/credentials.tfrc.json

Do you want to proceed?
  Only 'yes' will be accepted to confirm.

  Enter a value: yes
```

Generating and Storing your API Token

- During the CLI login process, HCP Terraform prompts you to create a new API token in the web interface.
- You can give the token a descriptive name or use the default.
- Once generated, copy the token and paste it into your terminal when prompted by `terraform login`.
- The CLI stores this token in a local credentials file for future authentication.



Ready for Remote Operations



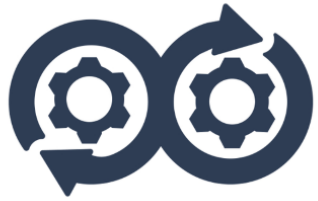
After successful authentication, your Terraform CLI is now connected to HCP Terraform. You can perform remote operations such as planning, applying, and managing state files, all within the secure and collaborative environment provided by HCP Terraform.

This setup allows you to leverage the benefits of cloud-based state management, team collaboration, and policy enforcement, while still using the familiar Terraform CLI workflow. With authentication complete, you are ready to create workspaces, manage variables, and provision infrastructure using HCP Terraform's robust cloud platform.

Ready for Remote Operations

```
Terraform must now open a web browser to the tokens page for app.terraform.io.

If a browser does not open this automatically, open the following URL to proceed:
    https://app.terraform.io/app/settings/tokens?source=terraform-login


------------------------------------------------------------------------------

Generate a token using your browser, and copy-paste it into this prompt.

Terraform will store the token in plain text in the following file
for use by subsequent commands:
    /Users/redacted/.terraform.d/credentials.tfrc.json

Token for app.terraform.io:
  Enter a value:


Retrieved token for user redacted

Welcome to HCP Terraform!
```

Lab: Authenticating to HCP Terraform



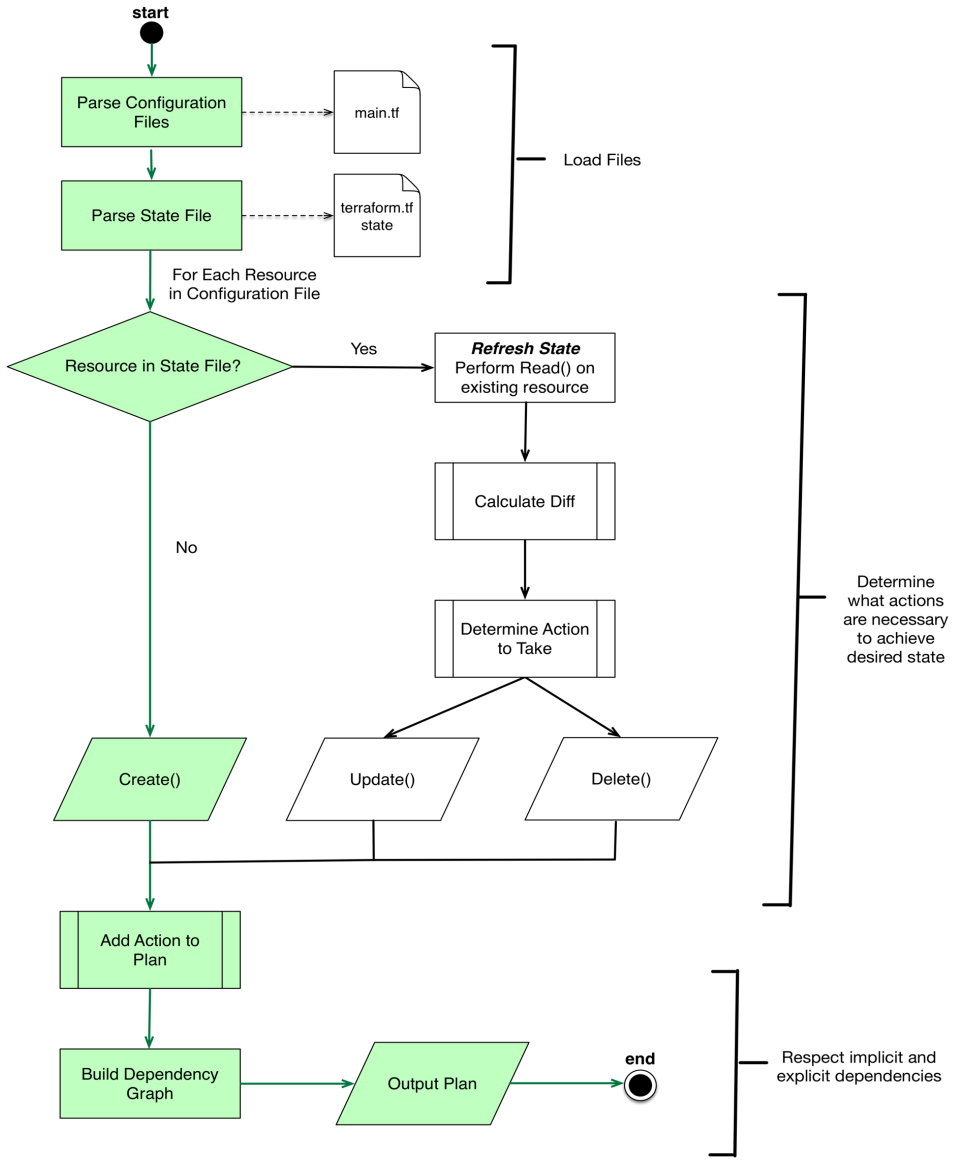
Terraform State Management

When Terraform creates a new infrastructure object represented by a resource block, the identifier for that real object is saved in Terraform's state, allowing it to be updated and destroyed in response to future changes.

Terraform State Management

Resource blocks that already have an associated infrastructure object in the state, Terraform compares the actual configuration of the object with the arguments given in the configuration and, if necessary, updates the object to match the configuration.

Terraform State Management

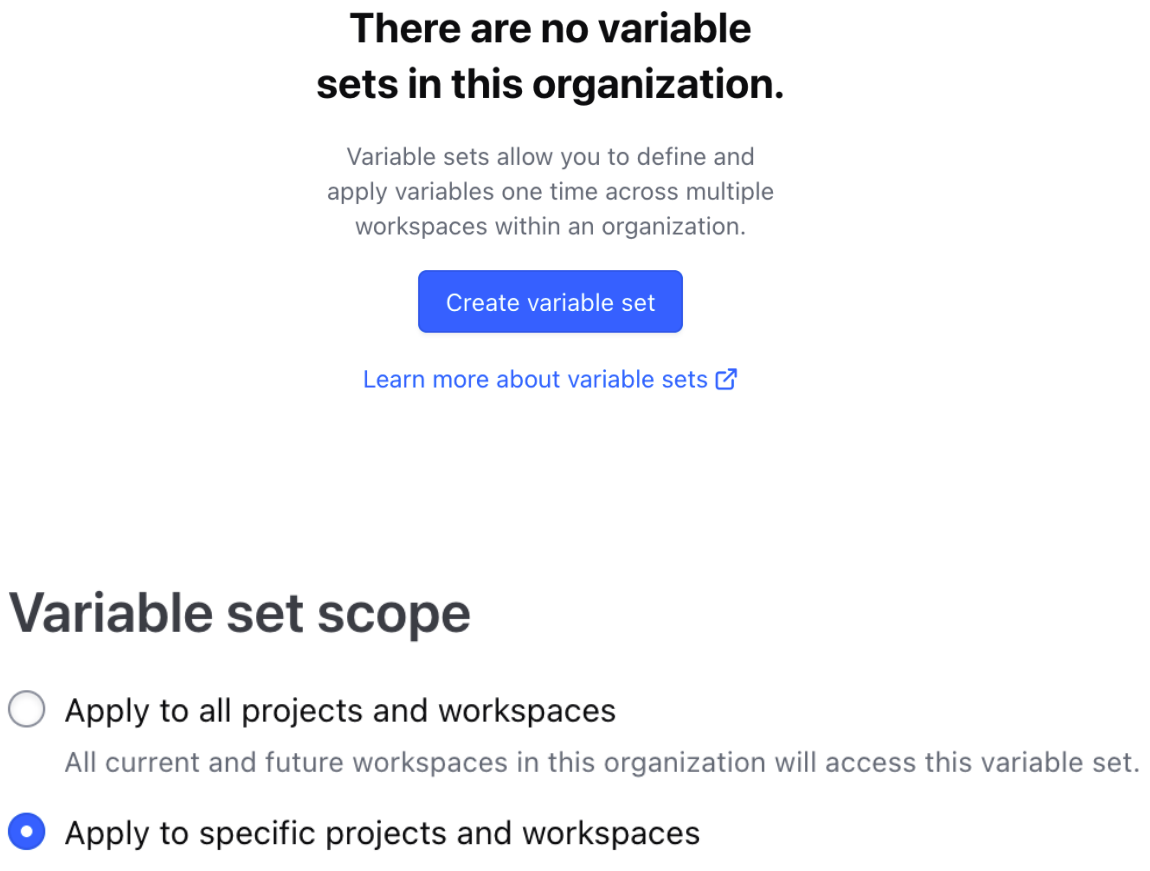


Introduction to Variable Sets

HCP Terraform allows you to manage input and environment variables centrally using variable sets. Variable sets help you avoid duplicating the same variables across multiple workspaces, making it easier to standardize and securely manage common configurations—such as provider credentials—throughout your organization.
By grouping variables into sets, you can efficiently rotate secrets and apply consistent settings to many workspaces at once, improving both security and operational efficiency.

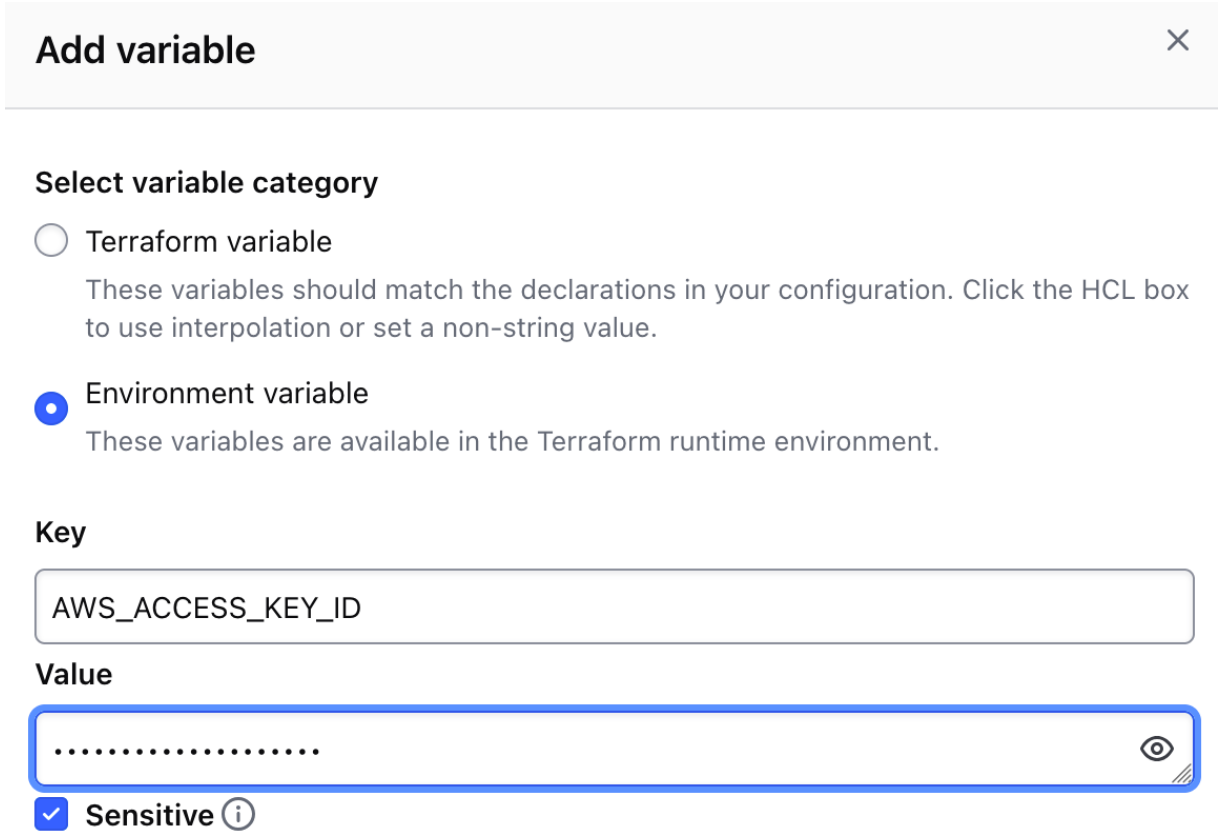
Creating a Variable Set

- Navigate to your organization’s Settings in the HCP Terraform UI, then select "Variable sets."
- Click "Create variable set" and give your set a descriptive name, such as "AWS Credentials."
- Choose whether to apply the variable set globally to all workspaces, or scope it to specific workspaces or projects for better access control.
- For credentials, it’s best practice to avoid global scope unless you are working in a dedicated tutorial or sandbox environment.



Adding and Securing Variables

- Click **"+ Add Variable"** to add each required variable to your set.
- Select "Environment variable" for provider credentials like `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
- Mark each credential as Sensitive to prevent it from being displayed in the UI.
- After adding all necessary variables, click "Create variable set" to save and make it available for assignment to workspaces.



HCP Terraform Workspaces Overview

A workspace in HCP Terraform is a logical grouping of infrastructure resources managed together as a unit. Each workspace contains everything Terraform needs to manage a specific collection of resources, including configuration, variables, state, and credentials.

Workspaces allow you to separate and organize your infrastructure, making it easier to manage different environments, projects, or components within your organization. Unlike local Terraform, where you might use directories to separate resources, HCP Terraform uses workspaces to provide isolation, access control, and collaboration features for your infrastructure as code workflows.

Workspace Contents and Features

- Stores Terraform configuration (from VCS or uploaded via API/CLI)
- Manages variable values and sensitive credentials within the workspace
- Maintains state files and retains backups of previous state versions
- Tracks run history, including logs, changes, and user comments
- Provides a resource count and visibility into managed resources
- Supports remote operations, running Terraform plans and applies on disposable virtual machines for security and consistency

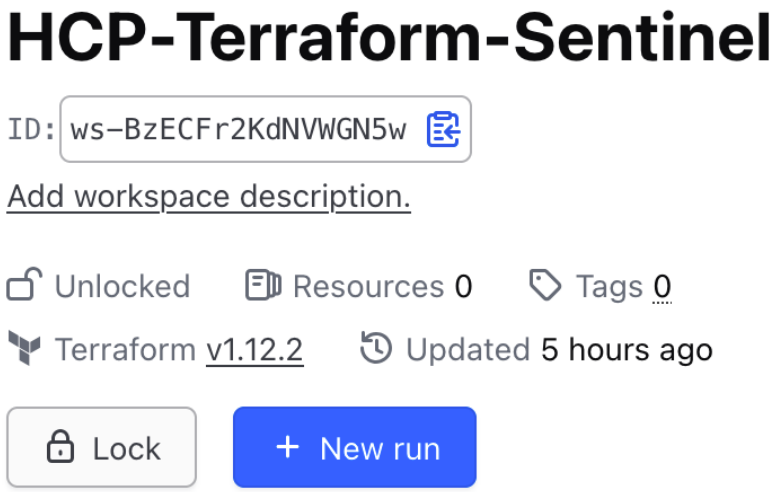
Creating a Workspace

```
terraform {
  cloud {
    organization = "your-
organization"
    workspaces {
      name = "your-workspace"
    }
  }
required_providers {
  aws = {
    source = "hashicorp/aws"
    version = "~> 5.31.0"
    }
  }

required_version = ">= 1.2"
}
```

To create a new workspace in HCP Terraform, you begin by updating your `terraform.tf` configuration file to include a `cloud` block. This block is essential because it tells Terraform which HCP Terraform organization and workspace to use for all operations performed in your current working directory. By specifying these details, you ensure that your local Terraform configuration is directly linked to a remote workspace in HCP Terraform, enabling centralized state management, collaboration, and access control.

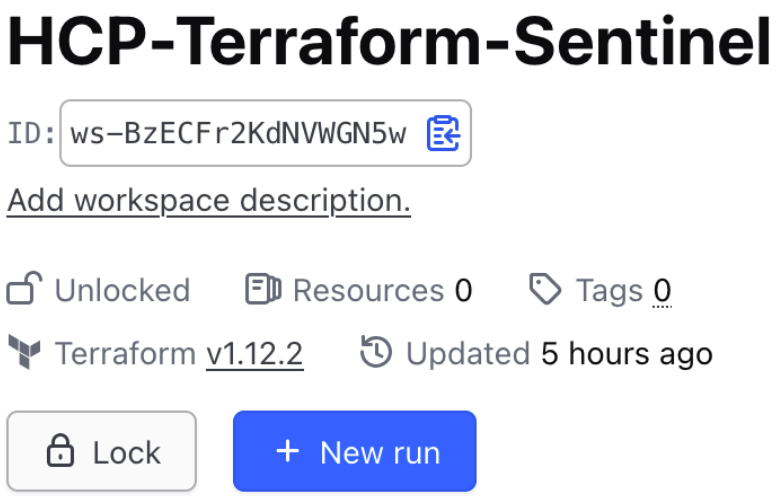
When you run `terraform init` with this configuration, Terraform will automatically create the specified workspace in your HCP Terraform organization if it does not already exist.

Initializing the Workspace with terraform init



- After updating your configuration, run `terraform init` in your project directory.
- Terraform will initialize the working directory, download required provider plugins, and create the new workspace in your HCP Terraform organization.
- The initialization process also creates a `.terraform.lock.hcl` file to record provider selections, ensuring consistent behavior across future runs.
- Once initialized, you can use commands like `terraform plan` and `terraform apply` to manage infrastructure through your new workspace.
- If you change modules or settings, rerun `terraform init` to reinitialize your directory.

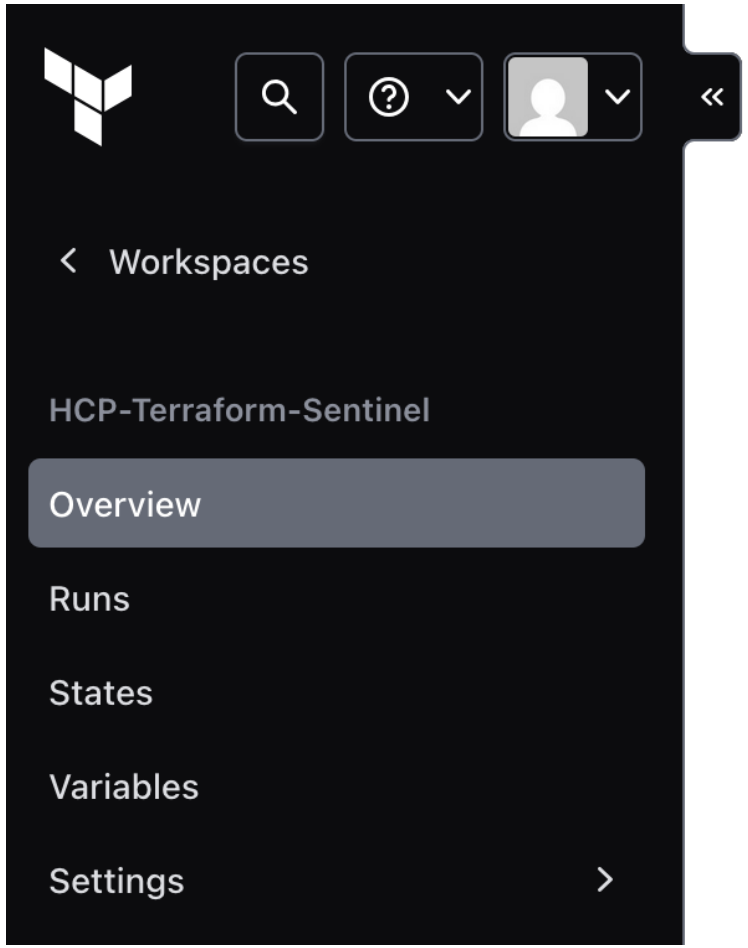
Exploring Your HCP Terraform Workspace



The workspace overview page in HCP Terraform provides a comprehensive snapshot of your current infrastructure state and configuration. At the top, you'll find key details such as the lock status, resource count, Terraform version, and the time since the last update.

The page also features controls for locking the workspace and triggering new runs. You can review the latest run details, including resource modifications, run duration, and estimated cost changes. Tables display all managed resources and outputs, while the sidebar offers quick access to workspace settings, tags, and metrics. This centralized view makes it easy to monitor and manage your infrastructure at a glance.

Navigating Workspace Actions and History



The workspace menu provides access to important actions and configuration pages:

- **Runs:** View a complete history of all plan and apply actions, with filtering by status, operation type, and source (UI, VCS, or API).
- **States:** Browse all past state files, which record the current and historical state of your resources.
- **Variables:** Manage Terraform variables, environment variables, and variable sets for the workspace.
- **Settings:** Adjust workspace settings or destroy infrastructure when needed.

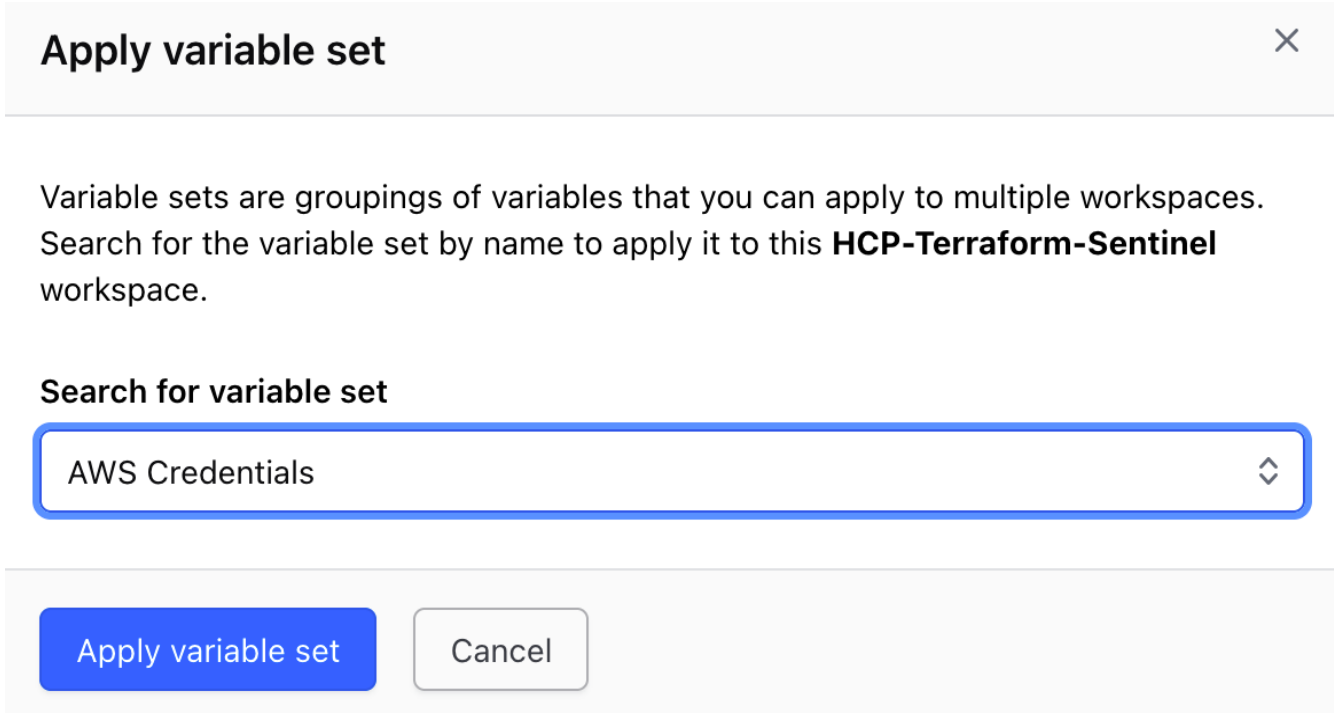
These features help you audit changes, troubleshoot issues, and maintain control over your infrastructure lifecycle.

Assigning Variable Sets to Your Workspace

Once your workspace is created, you may need to assign variable sets, such as AWS credentials to enable Terraform to provision resources.

- In the HCP Terraform UI, navigate to your workspace, select **"Variables,"** and under **"Variable sets,"** click **"Apply variable set."** Choose the appropriate variable set (e.g., **"AWS Credentials"**) and apply it.

This ensures your workspace has access to the necessary credentials and configuration values, and any updates to the variable set will automatically propagate to the workspace for future runs.

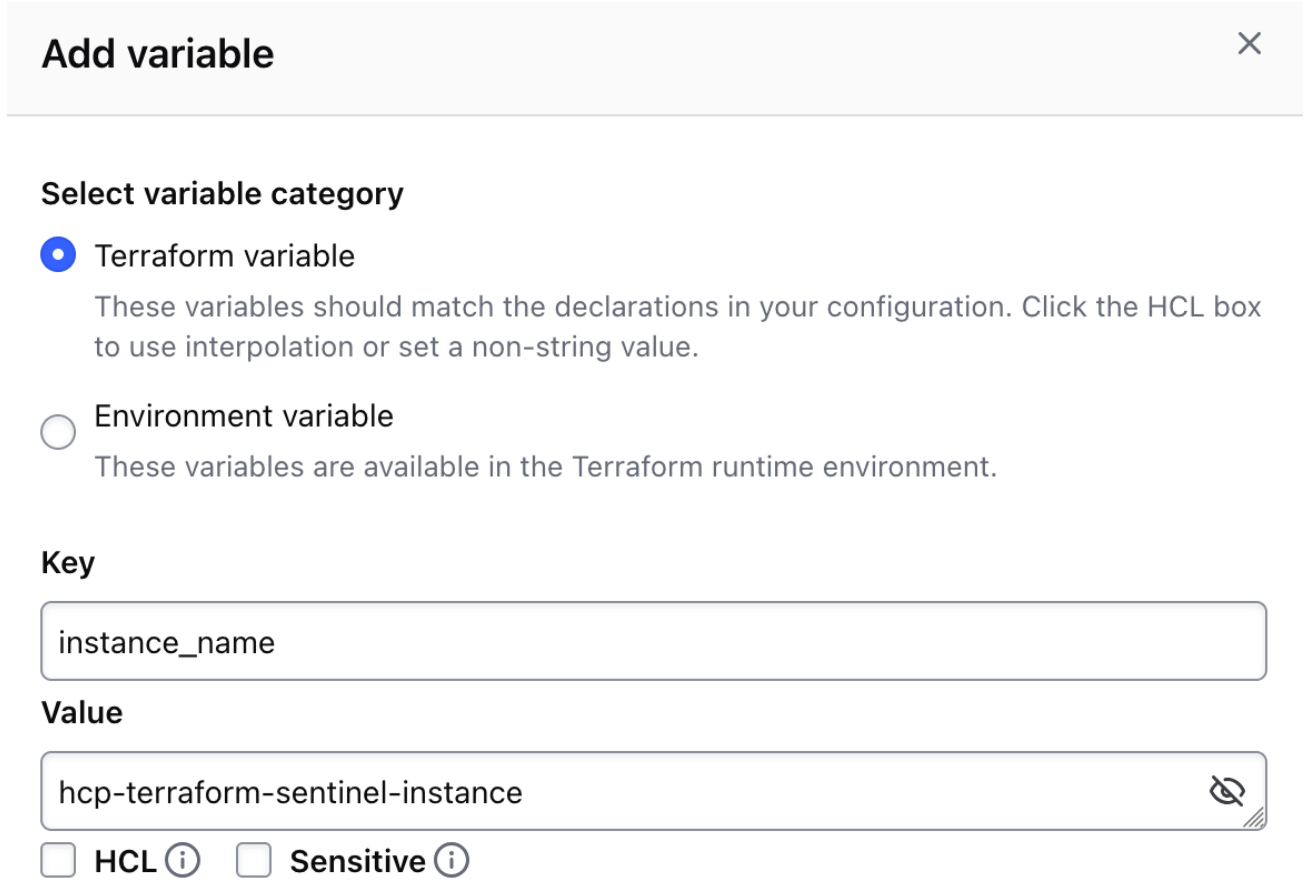


Configuring Workspace Variables

HCP Terraform allows you to define both input variables and environment variables directly in the workspace UI. To customize your infrastructure, navigate to the Variables page for your workspace.

- Here, you can add or modify variables such as **`instance_type`** or **`instance_name`** by selecting the **"Terraform variable"** option and entering the desired key and value.

These workspace-specific variables override defaults in your configuration and make it easy to adjust settings without editing code.



Applying Changes with terraform apply

Once your workspace variables are set, you can provision or update your infrastructure using the familiar **`terraform apply`** command.

When you run **`terraform apply`** in a CLI-driven workflow, Terraform streams the output of the remote run to your terminal and provides a link to view the run in the HCP Terraform UI. The plan step summarizes the proposed changes, giving you and your team a chance to review before anything is applied.

You must confirm and apply the plan, either from the terminal or directly in the UI, to make the changes take effect.

```
$ terraform apply
Running apply in HCP Terraform. Output will stream here. Pressing Ctrl-C
will cancel the remote apply if it's still pending. If the apply started it
will stop streaming the logs, but will not stop the apply running remotely.

Preparing the remote apply...
```

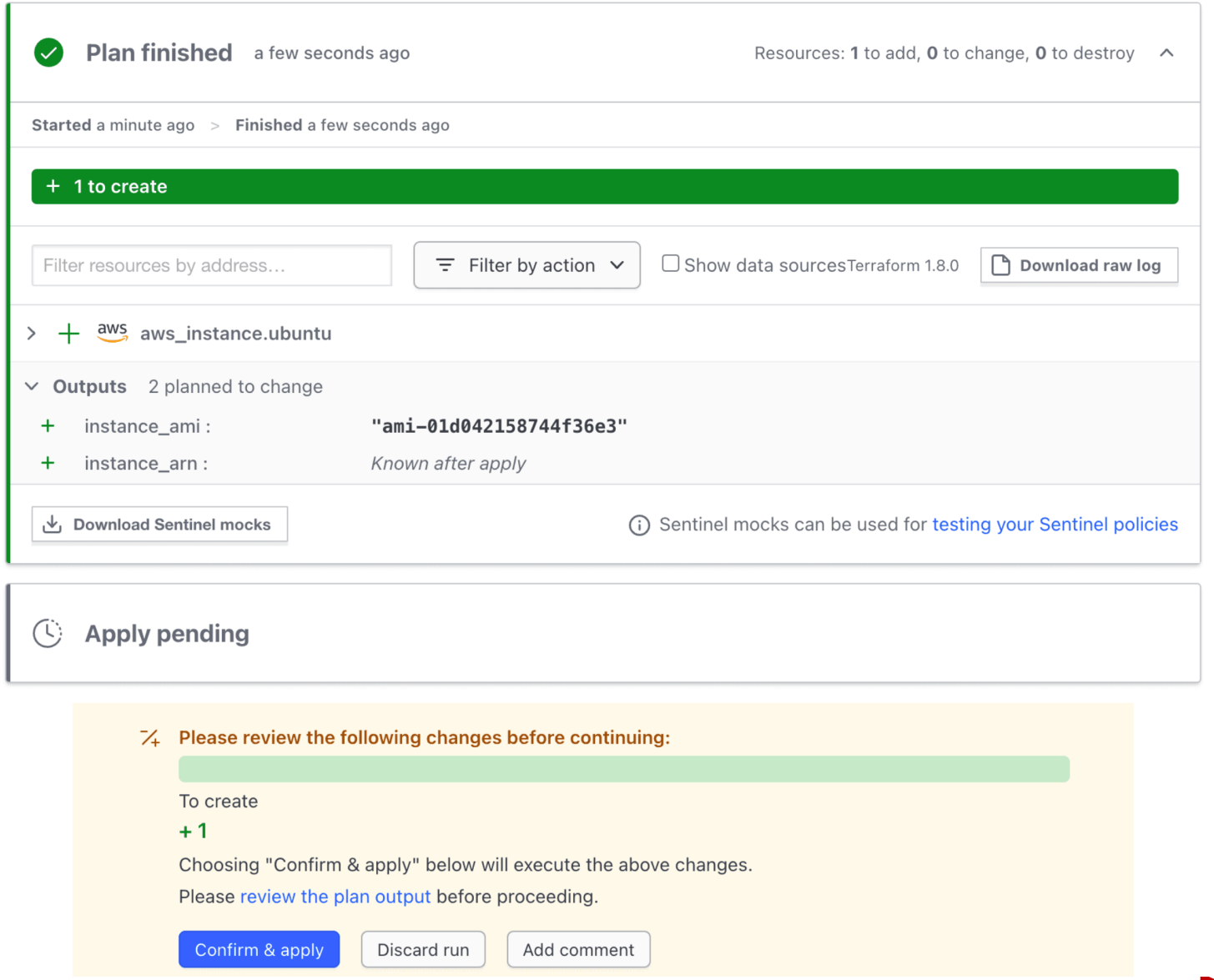
Editing Variables and Using Per-Run Overrides

HCP Terraform supports flexible variable management, allowing you to set variables in the UI or override them per run. You can pass variables on the command line using the `-var` flag, which temporarily overrides workspace-specific values for that run.

For example, running **`terraform apply -var="instance_type=t2.small"`** will update the instance type for your EC2 instance just for that operation. Note that per-run variables do not persist in the workspace, making them ideal for testing or temporary changes without altering the stored configuration.

Reviewing and Managing Runs

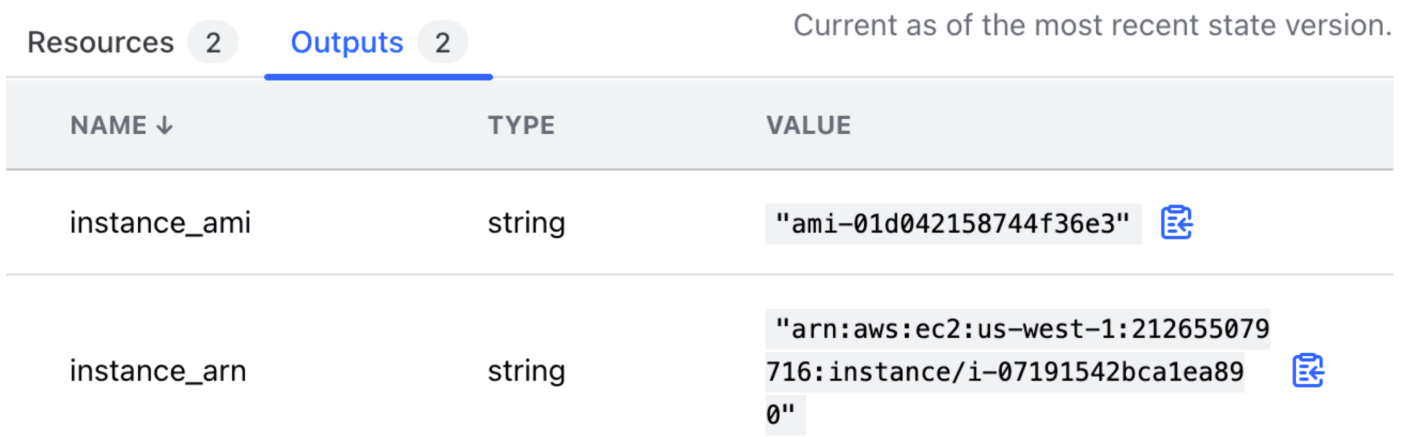
- Every Terraform run is tracked in the HCP Terraform workspace UI, providing a detailed history of all changes.
- You can view logs, plan details, and resource changes for each run, making it easy to audit and troubleshoot.
- The UI displays a table of all resources currently managed in the workspace, along with their status and outputs.
- This centralized visibility helps teams collaborate, review changes, and maintain compliance across all infrastructure operations.



Accessing Outputs and Verifying Infrastructure

After a successful apply, HCP Terraform displays the outputs defined in your configuration in the workspace's Outputs tab.

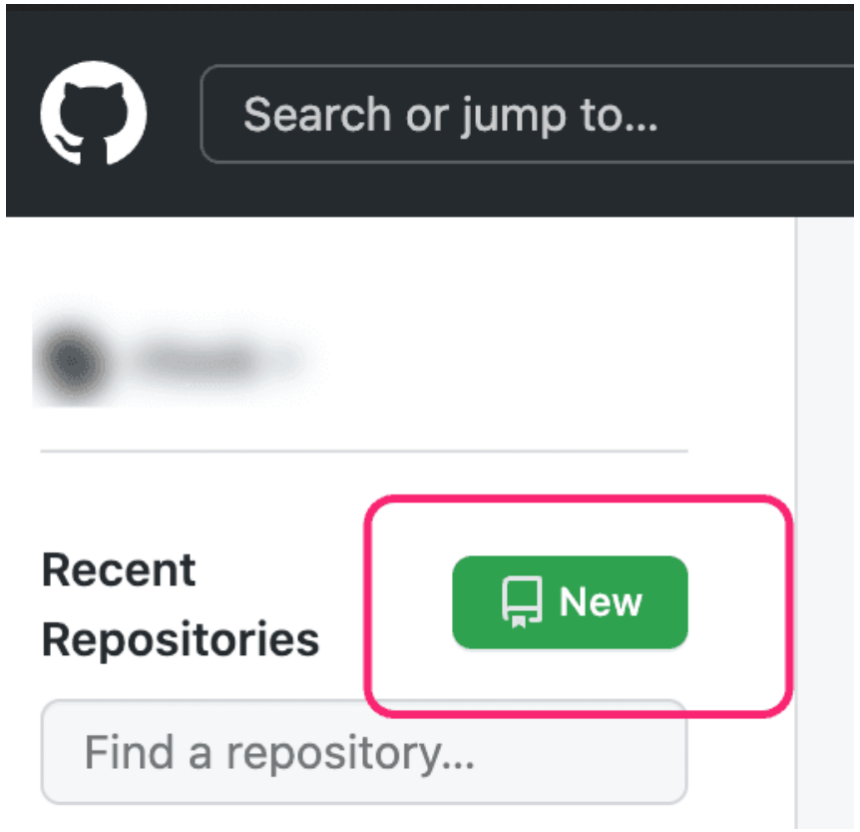
These outputs provide important information, such as resource IDs, IP addresses, or URLs, that you may need for further automation or integration. You can also verify that your infrastructure was created as expected by checking the relevant cloud provider console (e.g., AWS EC2 dashboard).



Introduction to the VCS-Driven Workflow

The VCS-driven workflow in HCP Terraform allows teams to manage infrastructure as code by connecting Terraform workspaces directly to version control repositories such as GitHub, GitLab, or Bitbucket. This workflow makes your repository the single source of truth for infrastructure configuration, enabling changes to be tracked, reviewed, and automatically applied through pull requests and merges.
By integrating with VCS, HCP Terraform streamlines collaboration, enforces best practices, and provides a clear audit trail for all infrastructure changes.

Configuring a New GitHub Repository



Create a new repository in your GitHub account to store your Terraform configuration files.
Copy the remote endpoint URL for your new repository.
Update your local git configuration to point to this new repository using `git remote set-url origin YOUR_REMOTE`.
This setup ensures all future changes are pushed to your personal or team-managed repository, supporting collaborative development.

Enabling VCS Integration in HCP Terraform

```
terraform {
/*
  cloud {
    organization = "organization-name"

    workspaces {
      name = "HCP-Terraform-Sentinel"
    }
  }
*/
}
```

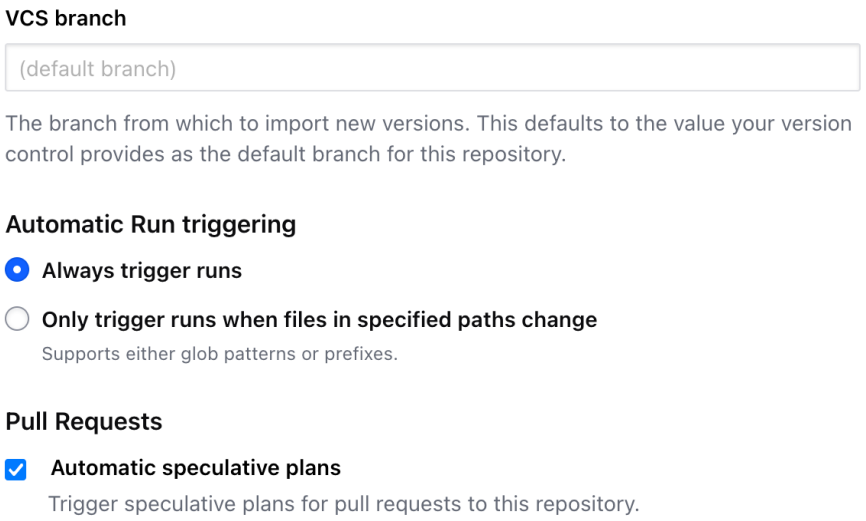
Before connecting your repository to HCP Terraform, update your Terraform configuration by commenting out or removing the `cloud` block in your `terraform.tf` file. The VCS-driven workflow does not require the `cloud` block, as workspace and organization settings are managed through the HCP Terraform UI. Commit and push these changes to your new repository to ensure your configuration is ready for integration with HCP Terraform.

Enabling VCS Integration in HCP Terraform

•In the HCP Terraform UI, go to your workspace’s settings and select the Version Control option.
•Click "Connect to version control" and choose your VCS provider (e.g., GitHub.com).
•Authorize HCP Terraform to access your repository and select the repository you created.
•Confirm the connection and enable automatic speculative plans, which preview infrastructure changes for every pull request.

- This setup allows HCP Terraform to automatically trigger runs based on changes in your repository.

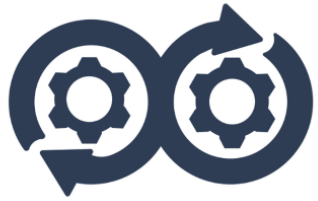
Automatic Runs and Speculative Plans



Once your workspace is connected to your VCS repository, HCP Terraform will automatically monitor for changes. Any push to the main branch of your repository will trigger a new Terraform run in HCP Terraform, ensuring that infrastructure changes are applied in a controlled and auditable manner. Additionally, when you open a pull request, HCP Terraform generates a speculative plan—a non-destructive, preview-only run that shows exactly what changes would be made if the pull request were merged.

This speculative plan allows your team to review, discuss, and approve proposed infrastructure modifications before they are actually applied, reducing the risk of errors and enabling safer collaboration. By leveraging automatic runs and speculative plans, teams can maintain a high level of confidence and transparency in their infrastructure workflows, catching issues early and ensuring that only reviewed changes reach production.

Lab: Modifying Infrastructure in HCP Terraform



Don't Repeat Yourself (DRY)

DRY is a principle that discourages repetition, and encourages modularization, abstraction, and code reuse. Applying it to Terraform, using modules is a big step in the right direction.

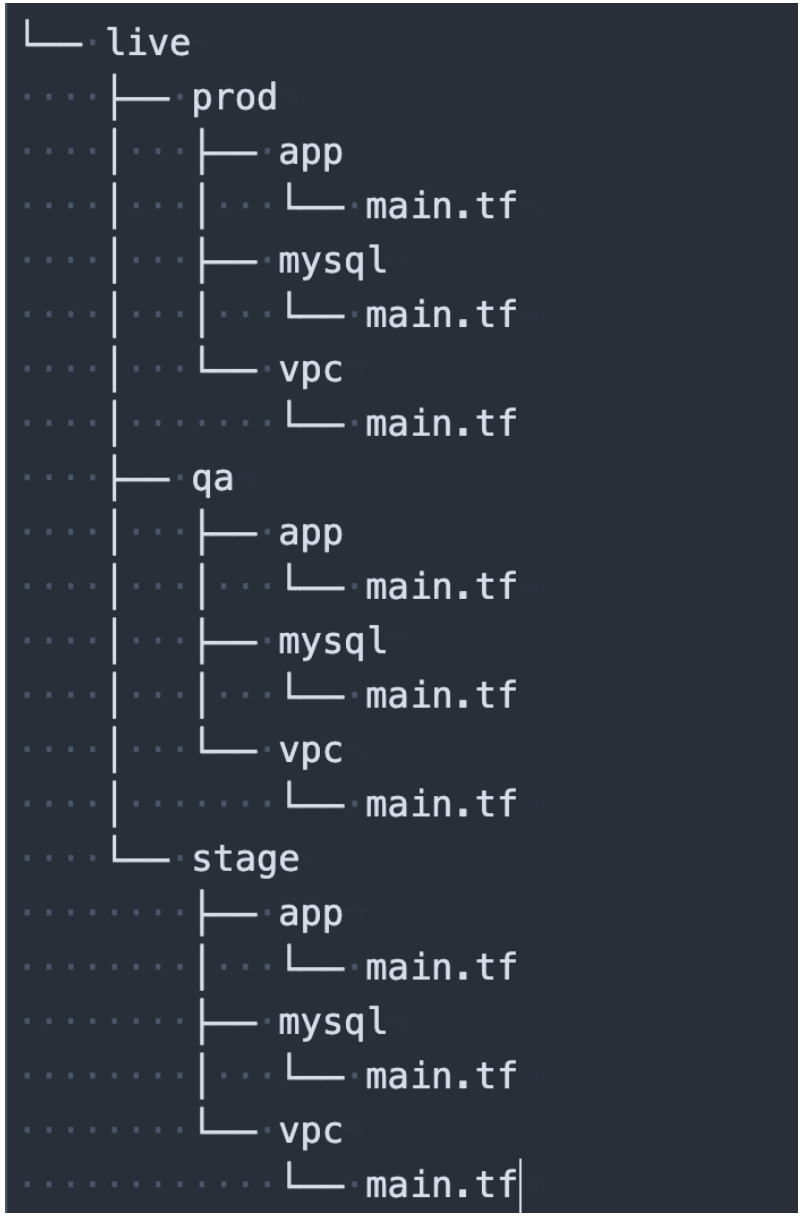
However, repetitions still happen. You may end up having virtually the same code in different environments, and when you need to make one change, you have to make that change many times.

Don't Repeat Yourself (DRY)

This problem can be addressed a few ways. One approach is to create a folder for shared or common files, and then create symlinks to these files from each environment. This way, you can make a change to the common file(s) once and it is applied in all the environments.

Don't Repeat Yourself (DRY)

Common directory structure for managing three environments (prod, qa, stage) with the same infrastructure (an app, a MySQL database and a VPC)



DRY Principle with Pipeline-Driven Environments

```
terraform/
├── main.tf              # Main configuration
├── variables.tf         # Variable definitions
├── outputs.tf           # Output definitions
├── terraform.tfvars     # Default values
└── environments/        # Environment-specific values
    ├── dev.tfvars
    ├── staging.tfvars
    └── prod.tfvars
```

While environment-specific directories are common, pipeline-driven environments with a single configuration provide better state management and control.

- Single configuration path for all environments
- Pipeline controls environment selection
- Automated state file management
- Reduced configuration duplication

Pipeline-Driven Environment Management

```
variables:
  TF_STATE_PREFIX:
${CI_PROJECT_NAME}/${CI_ENVIRONMENT_NAME}


init:
  script:
    - |
    # Configure backend for this environment
    cat > backend.hcl <<EOF
    bucket = "$TF_STATE_BUCKET"
    prefix = "$TF_STATE_PREFIX"
    EOF
    # Initialize with environment-specific
backend
    terraform init -backend-
config=backend.hcl
```

Using GitLab CI/CD pipelines to manage environments provides better control and state isolation.

- Pipeline selects environment configuration
- Automatic backend state path generation
- Environment-specific service accounts
- Controlled access through pipeline

Don't Repeat Yourself (DRY)

There is also an open source tool, Terragrunt which solves the same problem in a different way. It is a wrapper around the Terraform CLI commands, which allows you to write your Terraform once, and then in a separate repository define only input variables for each environment - no need to repeat Terraform code for each environment. Terragrunt is also handy for orchestrating Terrfaorm in CICD pipelines for multiple separate projects.

Standard Terraform Module Structure

Terraform modules are reusable collections of configuration files that follow a recommended structure.

The only required element is the root module, which consists of Terraform files in the root directory. Adhering to the standard structure is not mandatory, but it greatly improves documentation, usability, and compatibility with Terraform tooling. This structure also helps with automatic documentation generation and module registry indexing.

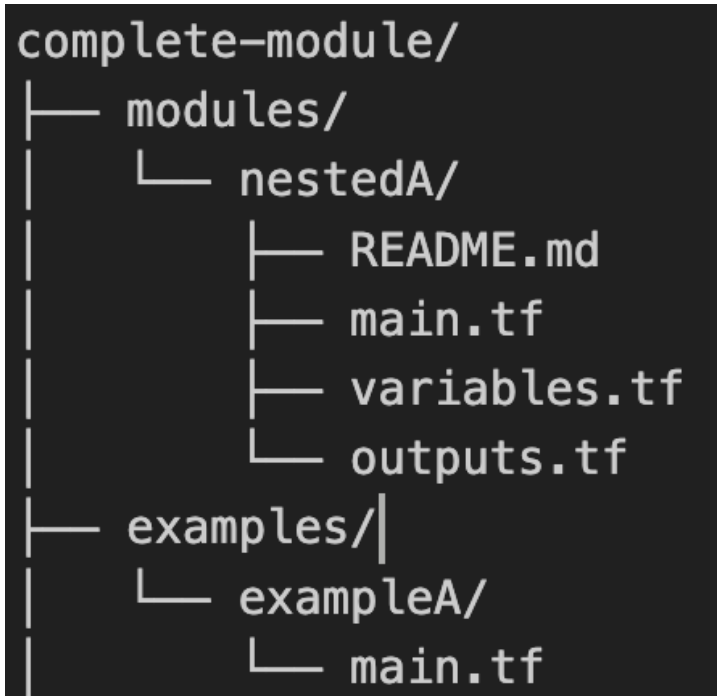
Minimal Recommended Module Layout

```
minimal-module/
├── README.md
├── main.tf
├── variables.tf
└── outputs.tf
```

A minimal Terraform module should include the following files to ensure clarity, reusability, and ease of use:

- README.md — Provides a description of the module, its purpose, and basic usage instructions.
- main.tf — The primary entrypoint for resource creation. This is where you define the main resources that the module manages.
- variables.tf — Declares all input variables for the module. Each variable should have a clear description, making it easier for users to know what values they need to provide.
- outputs.tf— Defines the outputs that the module will return. Outputs allow users to access information about resources created by the module, such as IDs or IP addresses.

Advanced Structure: Nested Modules



For more complex modules, you can organize your code with nested modules and usage examples.

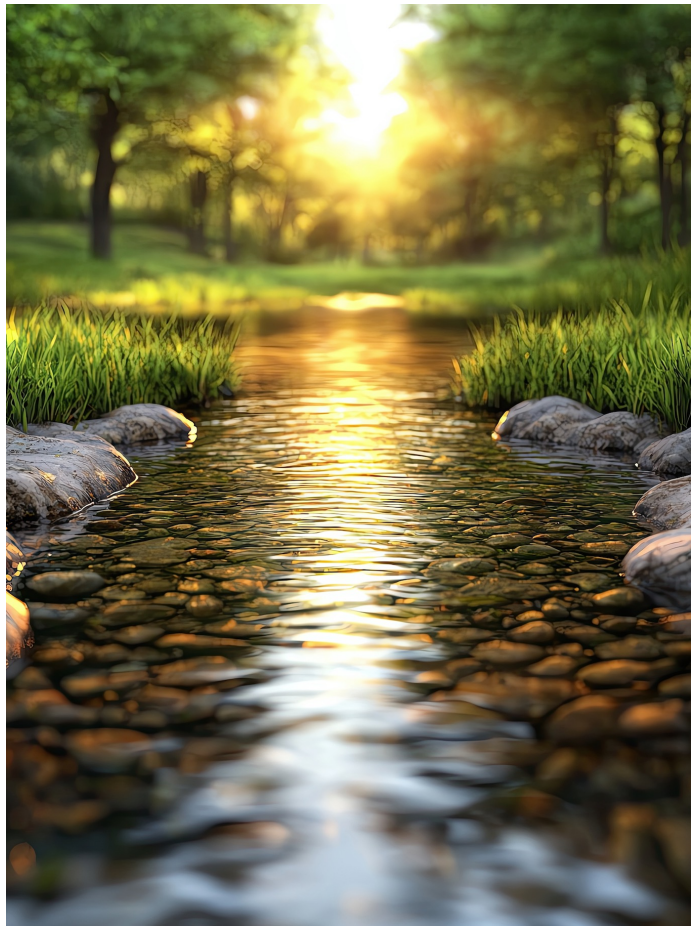
Nested modules are placed in a `modules/` subdirectory, each with its own configuration files and README. Usage examples should be placed in an `examples/` directory, with each example in its own folder. Including a LICENSE file is also recommended for public modules.

Terraform Module Design Overview

Terraform modules are self-contained, reusable pieces of infrastructure-as-code that help abstract complexity and promote best practices like DRY (Don't Repeat Yourself).

Good module design starts with scoping requirements into focused, opinionated modules that do one thing well. When designing a module, consider encapsulation (group resources that are always deployed together), privilege boundaries (keep resources with different access needs separate), and volatility (separate long-lived from short-lived infrastructure). Aim for a minimum viable product that covers most use cases, exposes useful outputs, and is well-documented. This approach makes modules easier to use, share, and maintain across teams and projects.

Use Source Control to Track Modules



A Terraform module should follow all good code practices, and source control is essential for this. Place each module in its own repository to manage release versions, enable collaboration, and maintain an audit trail of changes. Tag and document all releases to the main branch, using a CHANGELOG and README at minimum.

Code review all changes before merging to main, and encourage users to reference modules by tag for stability. Assign an owner to each module, and ensure only one module exists per repository—this supports idempotency, library-like usage, and is required for private registry compatibility.

Develop a Module Consumption Workflow

Define and publicize a repeatable workflow for teams consuming your modules. This workflow should be shaped by user requirements and make it easy for teams to adopt modules consistently. HCP Terraform offers tools like the private Terraform registry and configuration designer, which provide structure for module collaboration and consumption.

These tools simplify module discovery, usage, and integration, making modules more accessible and reducing onboarding friction for new users.

Make Modules Easy and Secure to Use

- Private Terraform registry: Offers a searchable, filterable way to manage and browse modules.
- UI: The Terraform Enterprise UI lowers the barrier for new users.
- Configuration designer: Provides interactive documentation and advanced autocompletion, helping users discover variables and outputs quickly.
- Devolved security: Repository RBAC allows teams to manage their own modules securely.
- Policy enforcement: Use Sentinel to enforce that only approved modules from the private registry are used, supporting compliance and governance.

Introduction To Publishing Private Modules

Publishing private modules to the HCP Terraform private registry enables organizations to securely share, version, and manage infrastructure modules.

The registry integrates with your version control system (VCS) and controls access, so module consumers do not need direct access to the source repository.

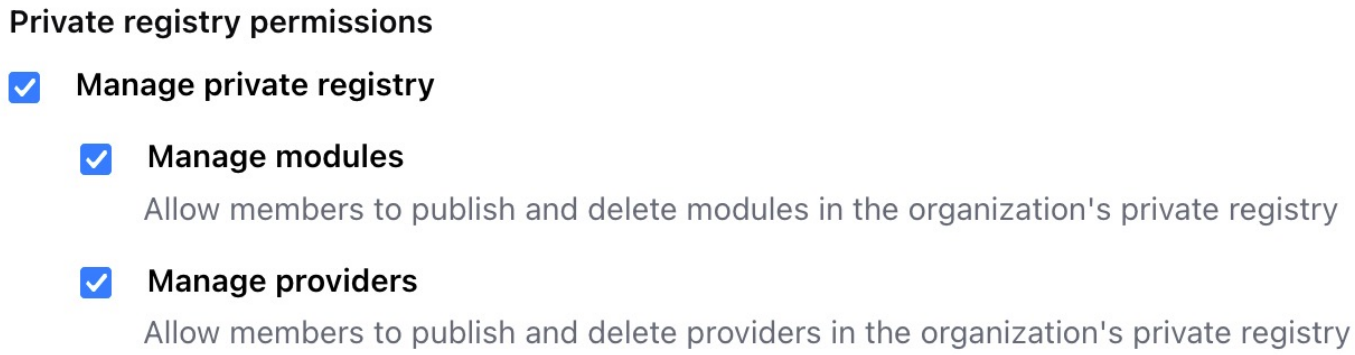
VCS Provider And Repository Requirements

Before publishing, ensure:

- Your VCS provider is connected to HCP Terraform.
- The module repository is accessible and follows the standard Terraform module structure.
- The registry user has admin access to the repository to set up webhooks and manage versions.

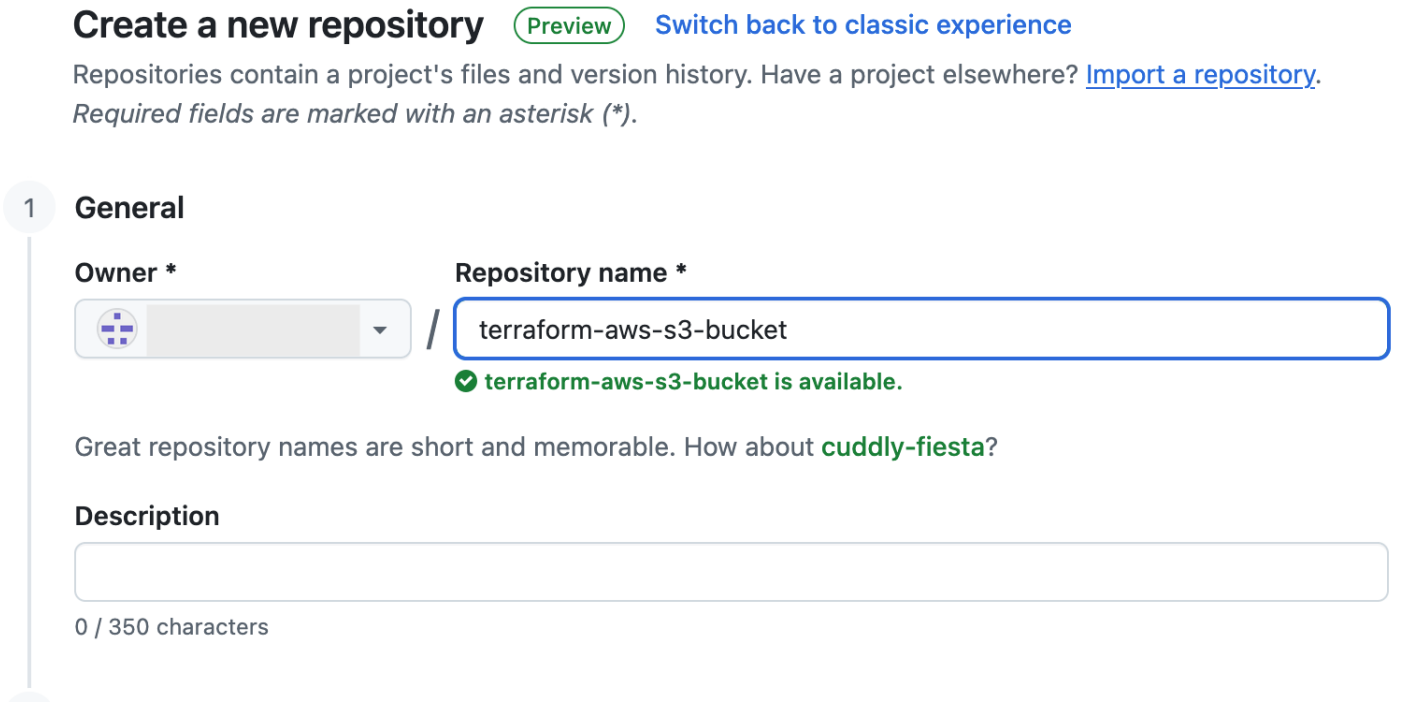
Private Registry Access And Permissions

- Only organization members with the "Manage private registry" permission or owner status can publish or delete modules.
- Private modules are visible only to members of the owning organization, unless sharing is explicitly configured with other organizations.



Module Naming Conventions

Each module should be stored in its own repository and follow the naming convention:
terraform-<PROVIDER>-<NAME>
For example, a module for AWS EC2 instances would be named `terraform-aws-s3-bucket`. This convention helps the registry identify and organize modules correctly.



Tag-based And Branch-based Publishing

You can publish modules using either tag-based or branch-based workflows.

- Tag-based publishing uses semantic version tags (e.g., v1.0.0) and is the default approach.
- Branch-based publishing allows you to publish from a specific branch and assign a version manually.

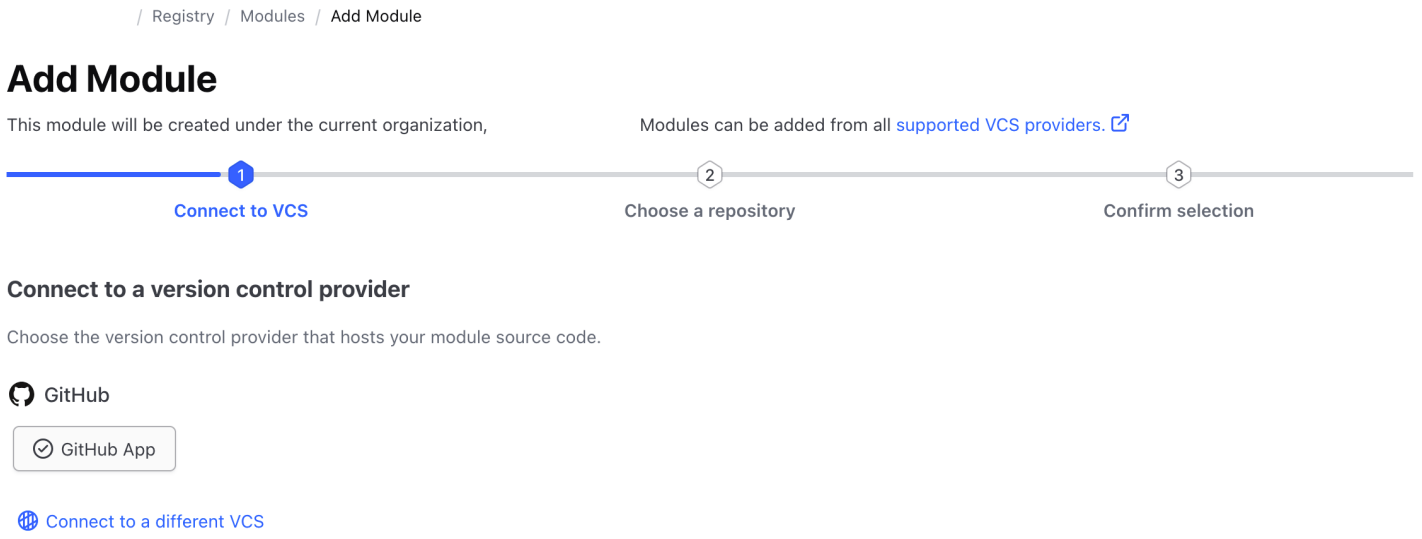
Choose the workflow that best fits your team's release process.

```
git tag v1.0.0
git push origin v1.0.0
```

Publishing A New Module

To publish a new module:

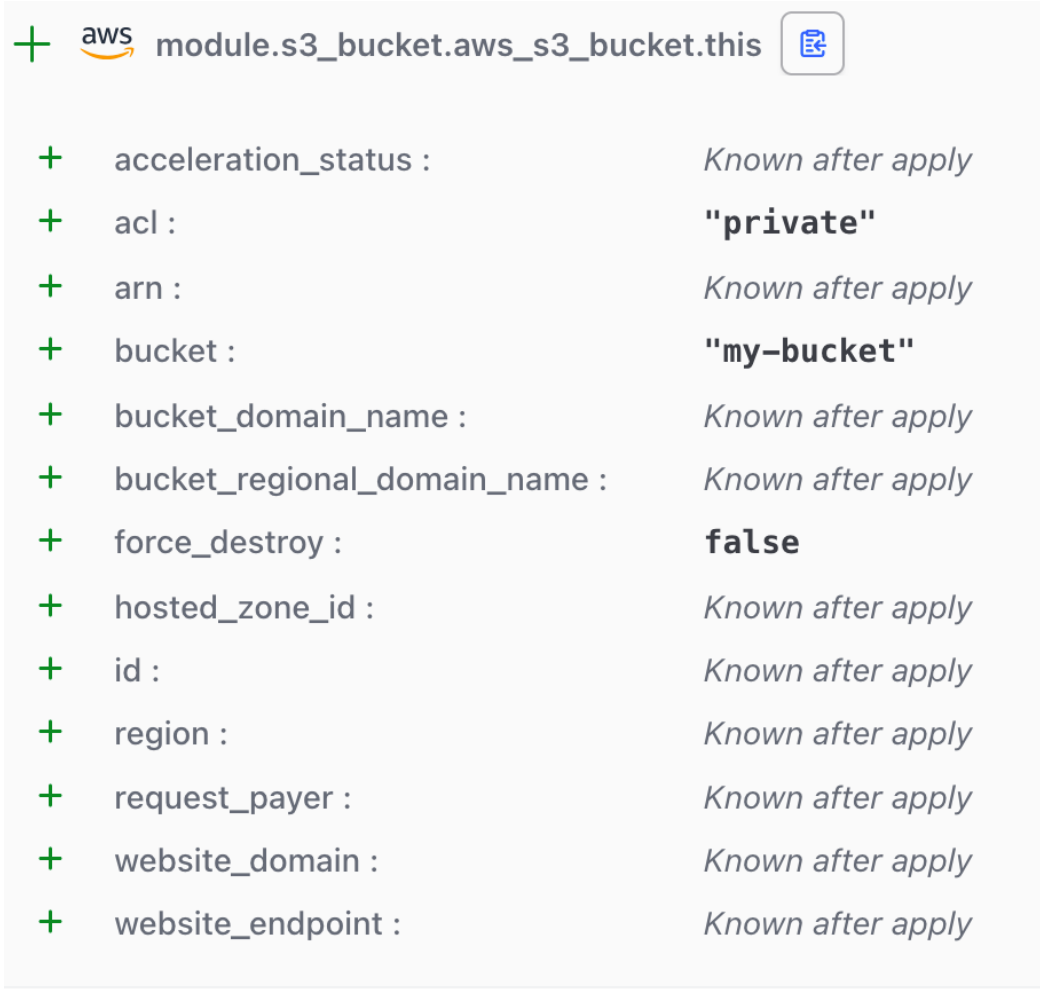
1. Go to the Registry in HCP Terraform and select "Module" from the Publish menu.
2. Choose your VCS connection and repository.
3. Configure the publishing type (tag or branch), specify the source directory, and fill in the module and provider names.
4. Complete the process to add the module to your private registry.



Releasing New Module Versions

For tag-based publishing, simply push a new semantic version tag to your VCS repository and the registry will automatically import the new version. This will trigger a new run in your HCP Terraform Workspace

For branch-based publishing, use the HCP UI to select a commit and assign a new version.



Managing Module Versions And Deletion

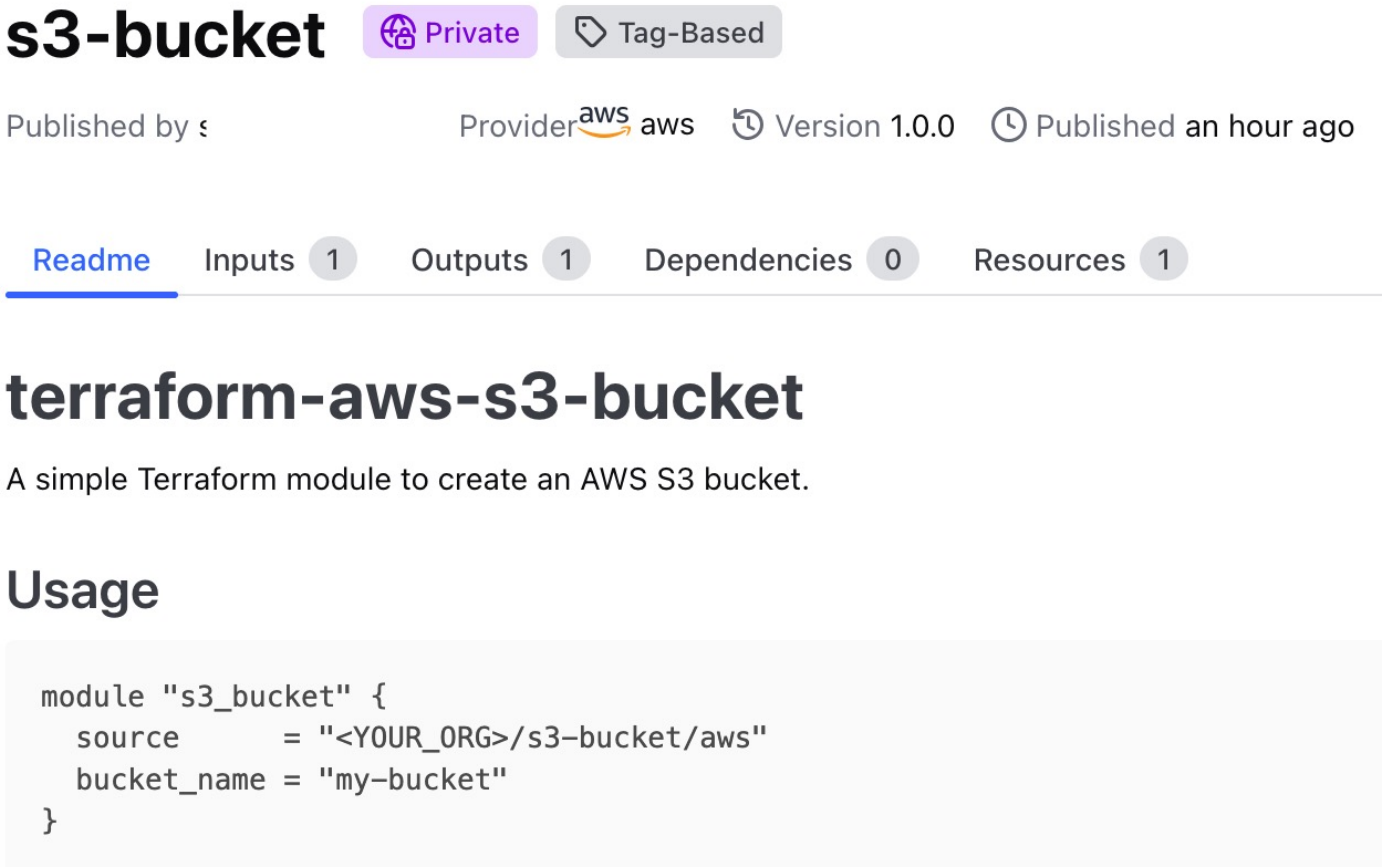
- You can delete individual module versions or entire modules from the registry.
- Deleting a tag in your VCS does not remove the version from the registry; you must delete it in the registry UI.
- If you delete the last version, the module is removed entirely.
- You can delete a module using the API

```
curl -k --header "Authorization: Bearer $TOKEN" \
--header "Content-Type: application/vnd.api+json" \
--request DELETE \
https://$TFE_URL/api/v2/organizations/<ORG_NAME>/\
registry-modules/private/<ORG_NAME>/<MODULE_NAME>
```

Sharing Modules Across Organizations

Sharing modules across organizations is possible by sharing the underlying VCS repository and adding the module to each organization's registry.

In Terraform Enterprise, module sharing can be configured for more advanced scenarios.

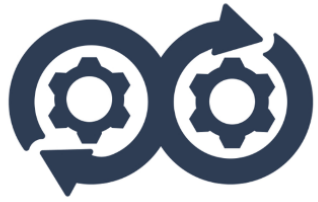


Best Practices And Next Steps

- Use one module per repository.
- Tag and document all releases.
- Assign an owner to each module.
- Manage access and permissions carefully.
- Leverage the registry’s features to track, test, and share modules efficiently.

For more details, refer to the official documentation and workflows.

Lab: Publishing a Terraform Module



Multi-Team Infrastructure Coordination with HCP Terraform

HCP Terraform provides powerful features for coordinating infrastructure changes across multiple teams while maintaining security, compliance, and operational stability.

Through workspace organization, variable sets, team permissions, and policy enforcement, organizations can establish clear boundaries and workflows that enable teams to work independently while ensuring infrastructure changes are properly reviewed and audited.

Workspace Organization and Team Structure

Organize workspaces to reflect your team structure and infrastructure boundaries:

- Use the recommended naming convention: `<business-unit>-<app-name>-<layer>-<env>`
- Example: `network-team-webapp-compute-prod` identifies ownership, application, layer, and environment
- Create separate workspaces for different teams and responsibilities
- Assign appropriate team permissions to each workspace based on ownership

This structure allows teams to work independently while maintaining clear ownership and accountability.

Grouping by Volatility and State Management

Group resources by their rate of change to minimize blast radius:

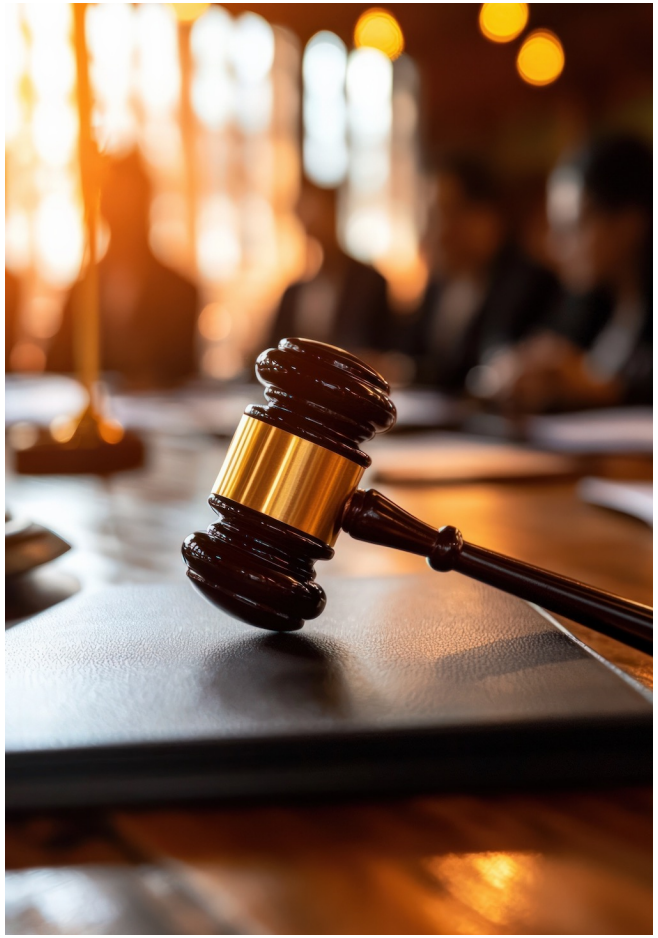
- Long-lived infrastructure (databases, VPCs, subnets) in separate workspaces from frequently changing resources
- Separate stateful resources (databases, object storage) from stateless ones (compute instances)
- Protect against accidental data loss during resource recreation
- Reduce the risk of exposing stable infrastructure to unnecessary changes

Project-Based Access Control and Variable Sets

Use projects to group related workspaces and scope team access appropriately. Create project-level variable sets for shared configuration like cloud provider credentials, default tags, and cost codes.

This approach enables least-privilege access while reducing duplication and ensuring teams use consistent, approved values for their infrastructure.

Policy Enforcement and Cross-Team Coordination



Implement Sentinel policies across projects and workspaces to enforce organizational standards automatically. Use remote state sharing through workspace outputs and the tfe_outputs data source to enable teams to reference resources from other workspaces without exposing sensitive information.

This approach allows teams to build upon each other's work while maintaining clear boundaries and reducing coordination overhead.