Terraform Core & Beyond

TEKsystems Global Services

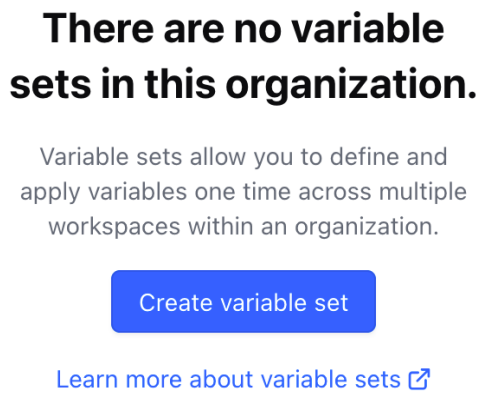
WORKFORCE DEVELOPMENT

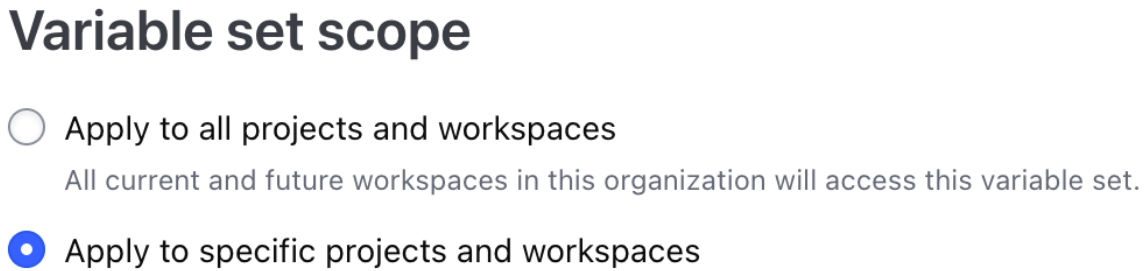
Introduction to Variable Sets

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

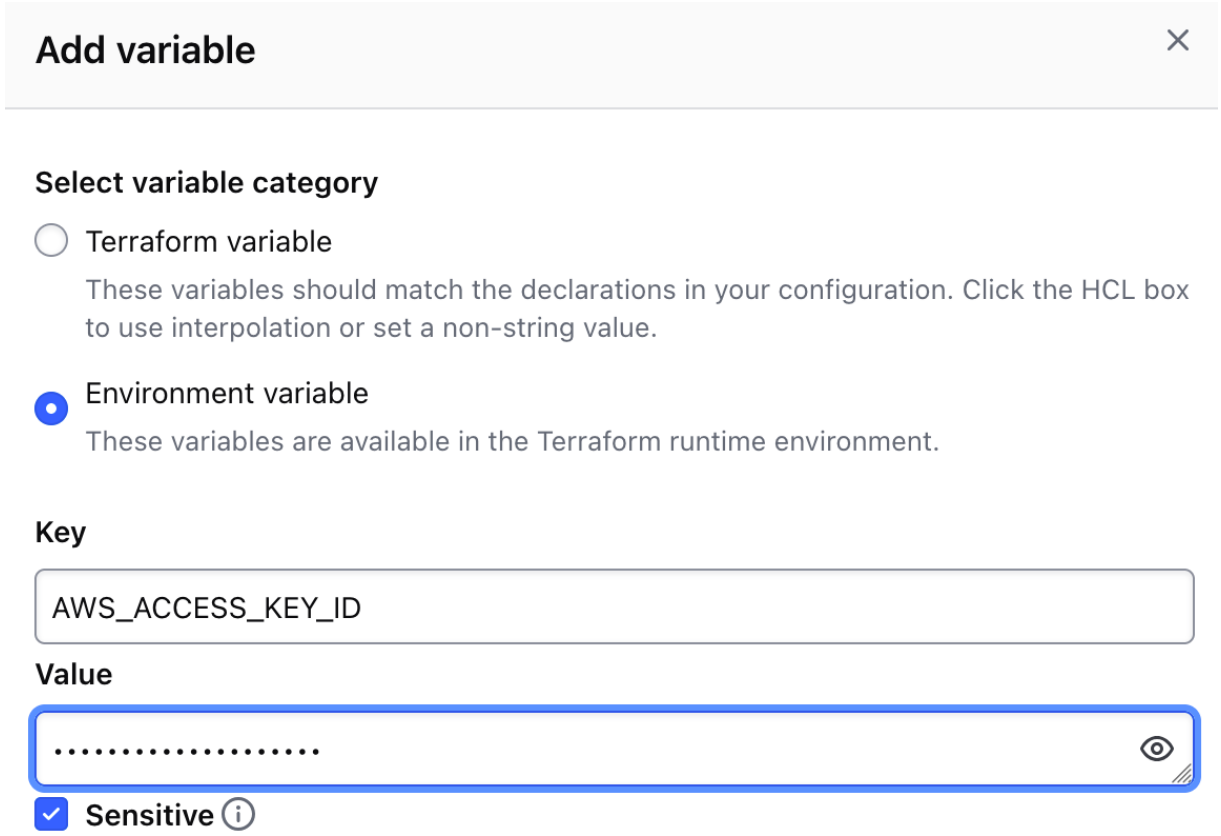
- Navigate to your organization’s Settings in the HCP Terraform UI, then select "Variable sets."
- Click "Create variable set" and give your set a descriptive name, such as "AWS Credentials."
- Choose whether to apply the variable set globally to all workspaces, or scope it to specific workspaces or 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

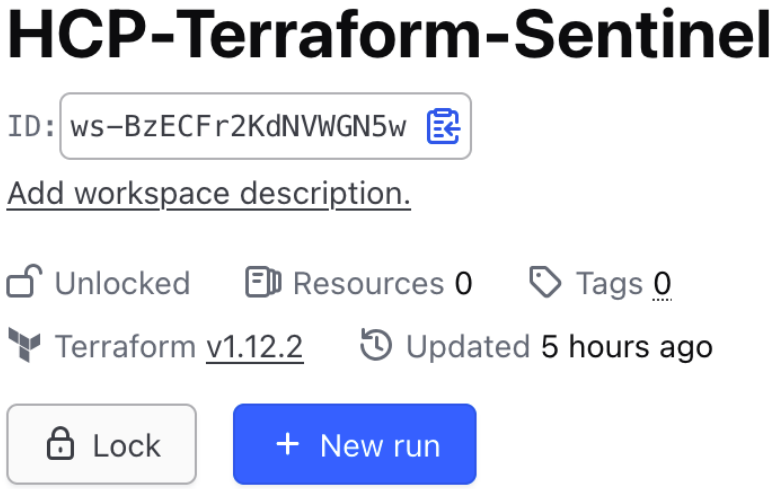
```
terraform {
  cloud {
    organization = "your-
organization"
    workspaces {
      name = "your-workspace"
    }
  }
required_providers {
  aws = {
    source = "hashicorp/aws"
    version = "~> 5.31.0"
    }
  }

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.

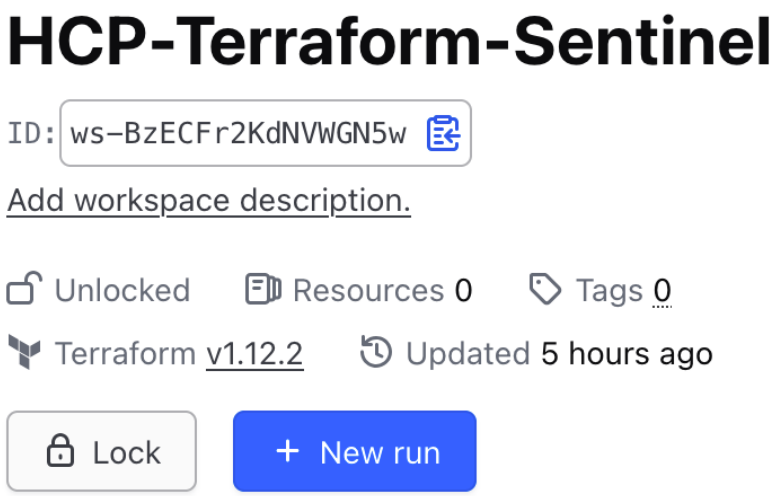
When you run `terraform init` with this configuration, Terraform will automatically create the specified workspace in your HCP Terraform organization if it does not already exist.

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.

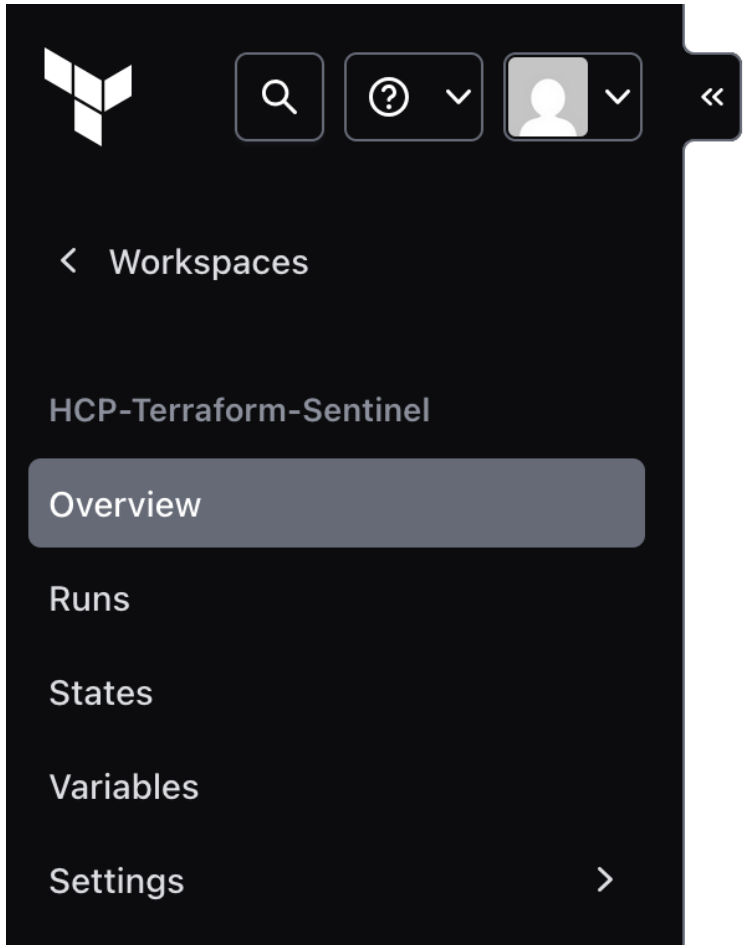
Exploring Your HCP Terraform Workspace



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.

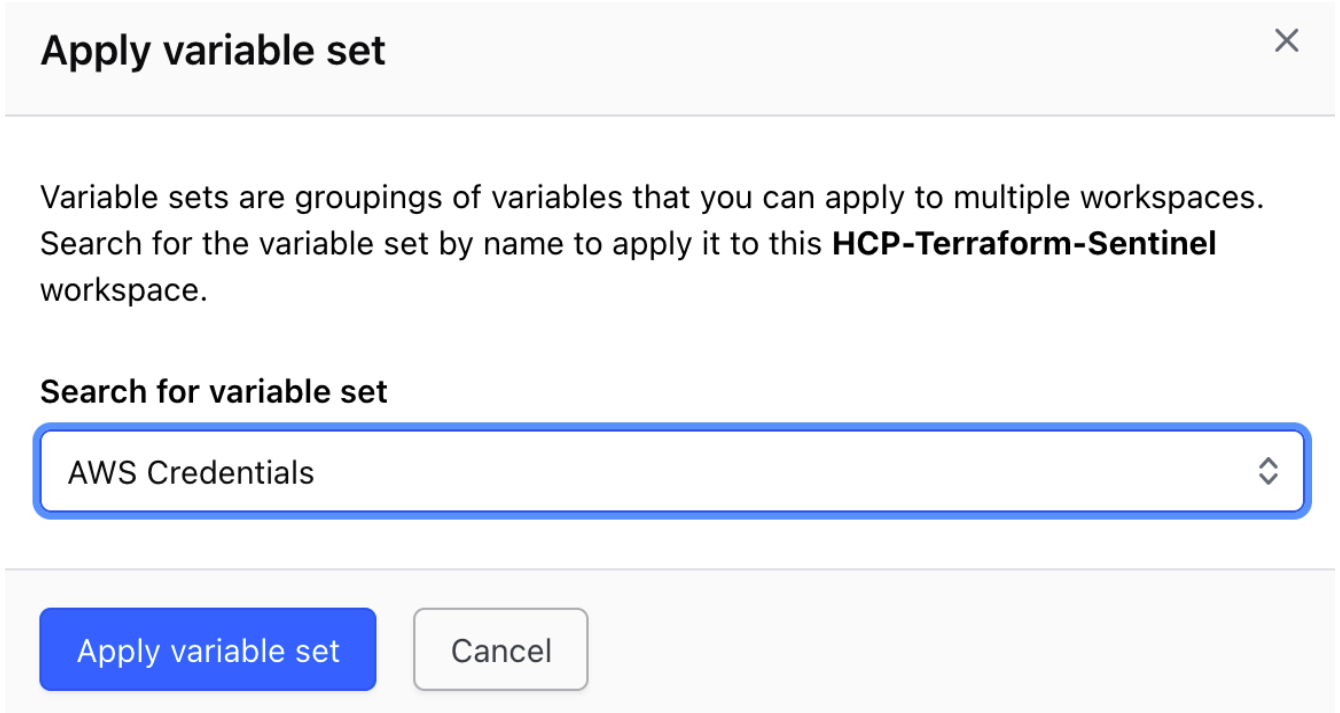
These features help you audit changes, troubleshoot issues, and maintain control over your infrastructure lifecycle.

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.

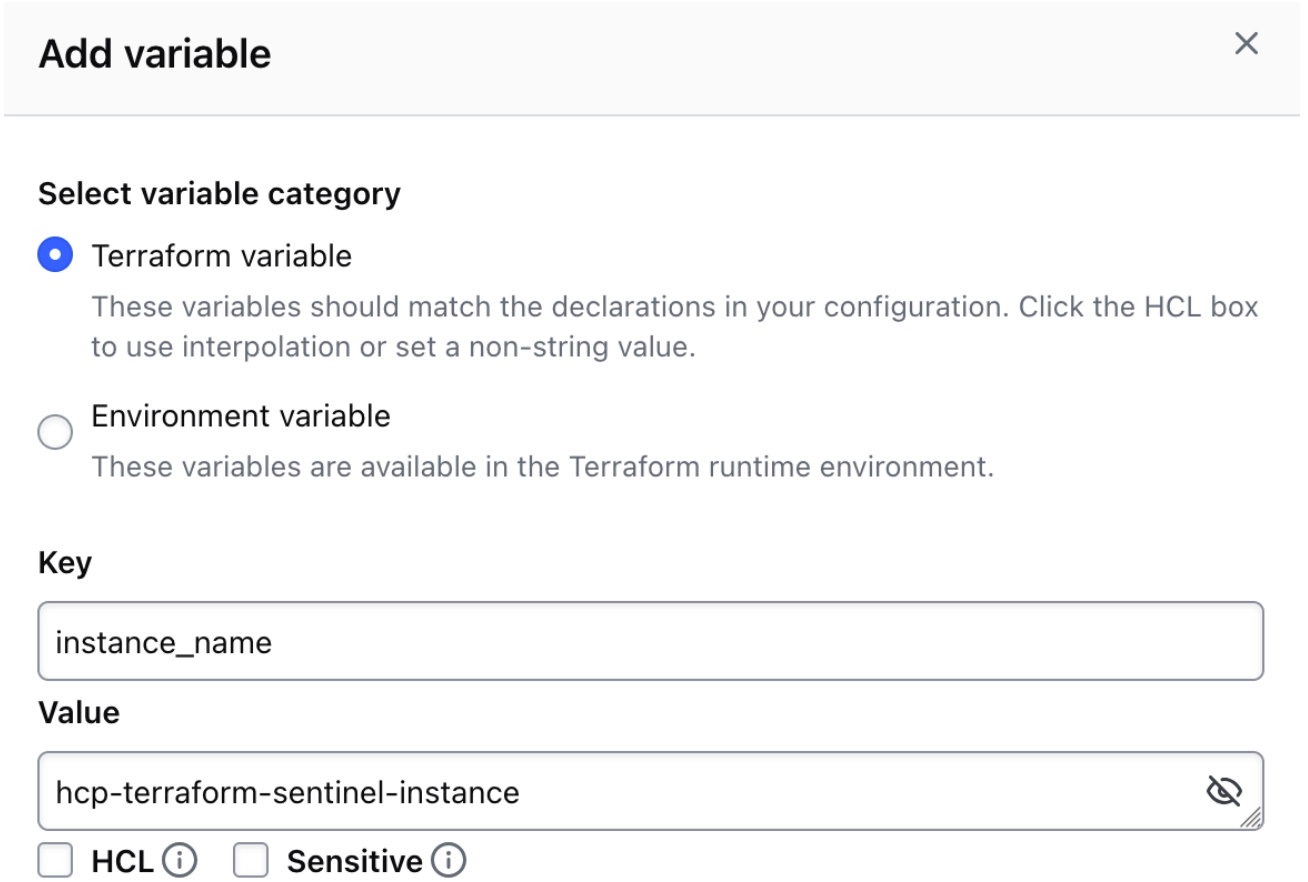


Configuring Workspace Variables

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...
```

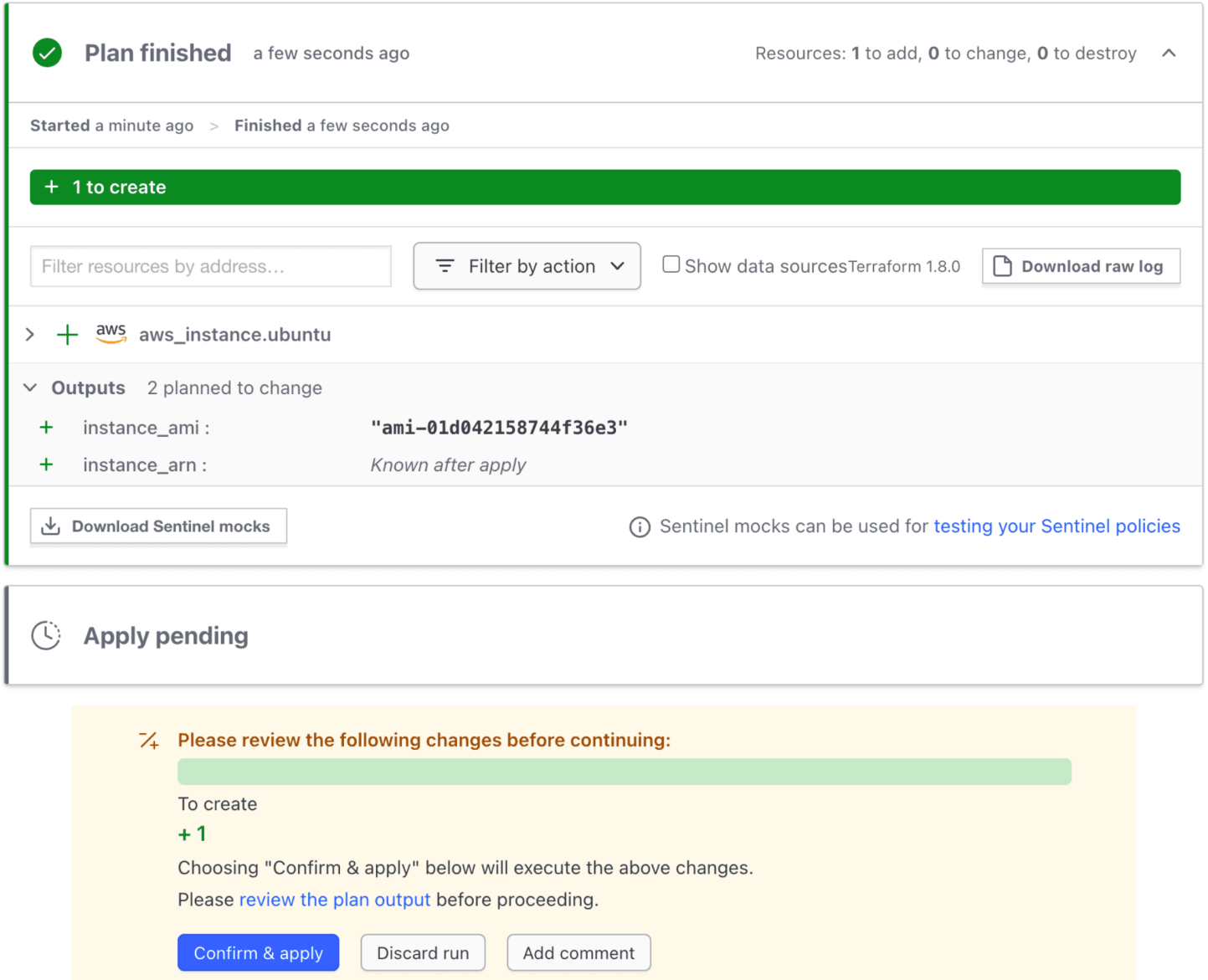
Editing Variables and Using Per-Run Overrides

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

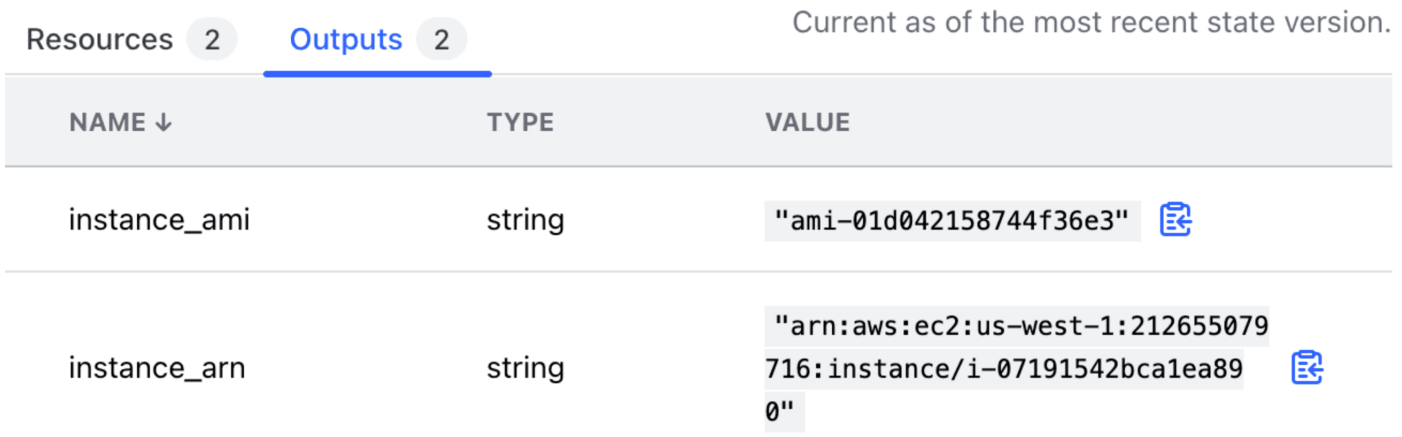
- Every Terraform run is tracked in the HCP Terraform workspace UI, providing a detailed history of all changes.
- You can view logs, plan details, and resource changes for each run, making it easy to audit and troubleshoot.
- The UI displays a table of all resources currently managed in the workspace, 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.

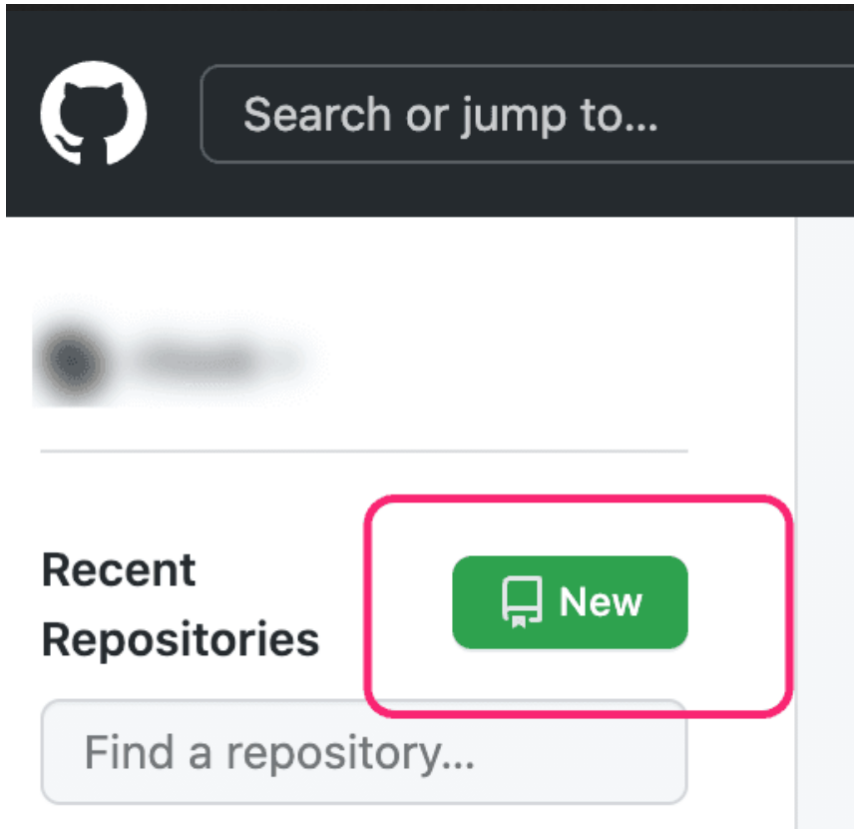
These outputs provide important information, such as resource IDs, IP addresses, or URLs, that you may need for further automation or integration. You can also verify that your infrastructure was created as expected by checking the relevant cloud provider console (e.g., AWS EC2 dashboard).



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"
    }
  }
*/
}
```

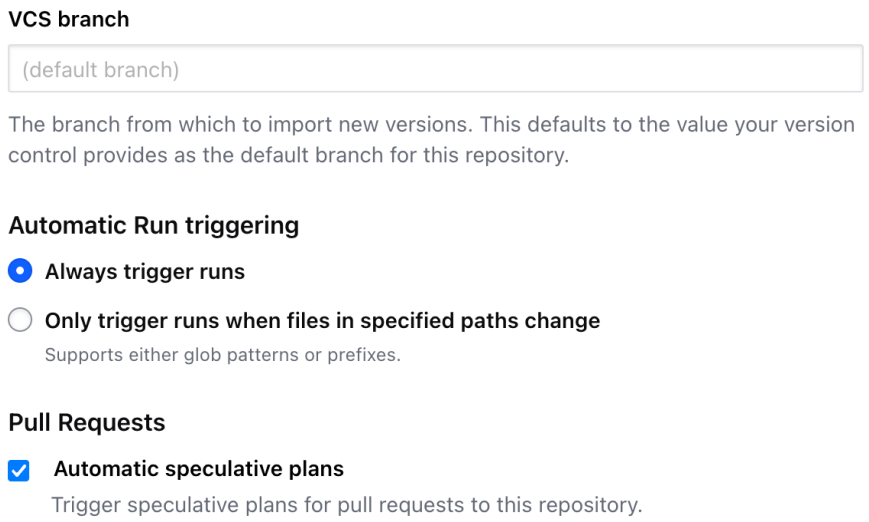
Before connecting your repository to HCP Terraform, update your Terraform configuration by commenting out or removing the `cloud` block in your `terraform.tf` file. The VCS-driven workflow does not require the `cloud` block, as workspace and organization settings are managed through the HCP Terraform 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.

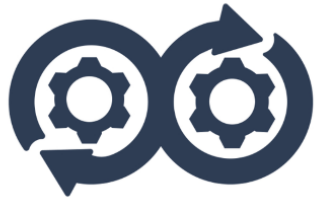
Automatic Runs and Speculative Plans



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.

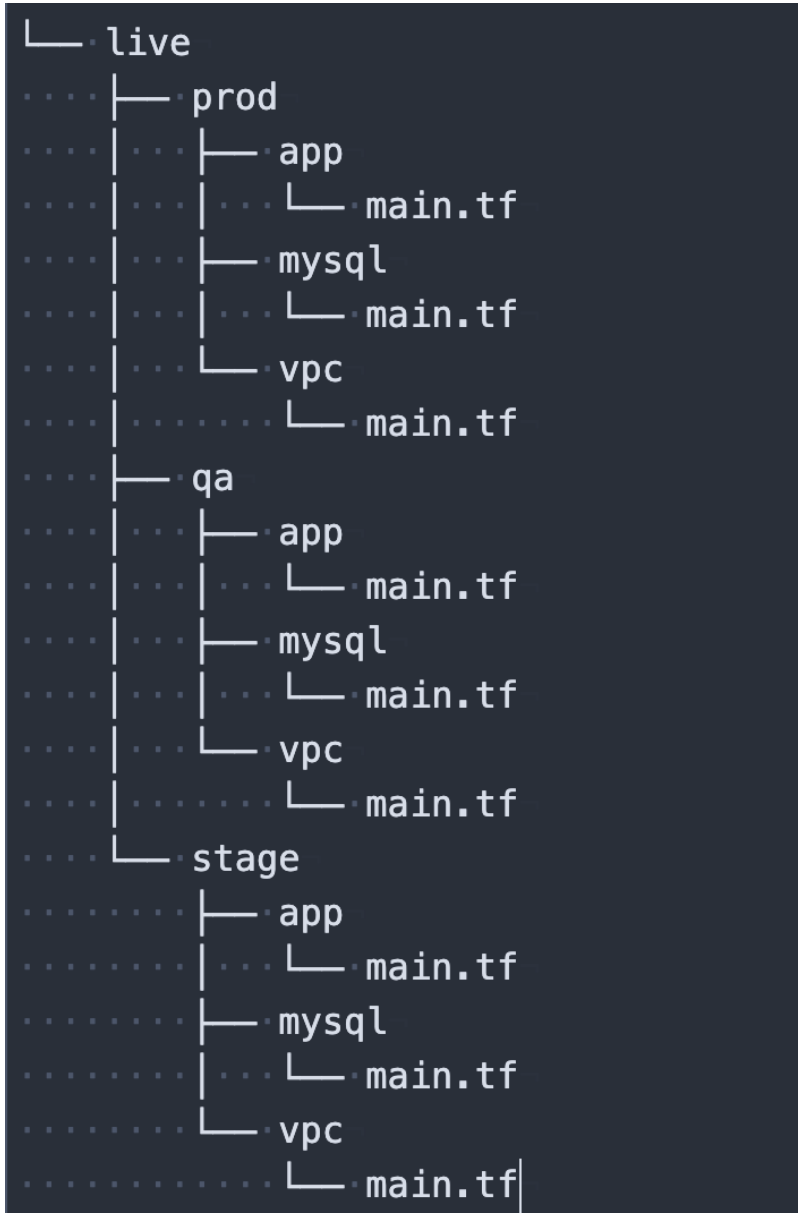
However, repetitions still happen. You may end up having virtually the same code in different environments, and when you need to make one change, you have to make that change many times.

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)

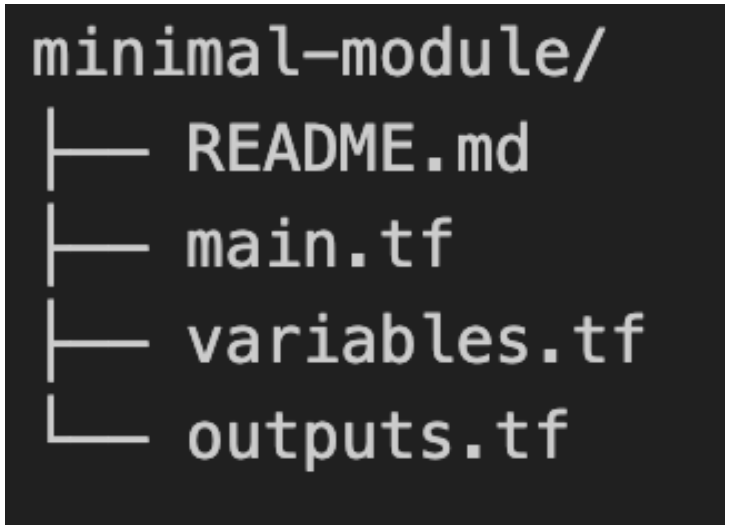
There is also an open source tool, Terragrunt which solves the same problem in a different way. It is a wrapper around the Terraform CLI commands, which allows you to write your Terraform once, and then in a separate repository define only input variables for each environment - no need to repeat Terraform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

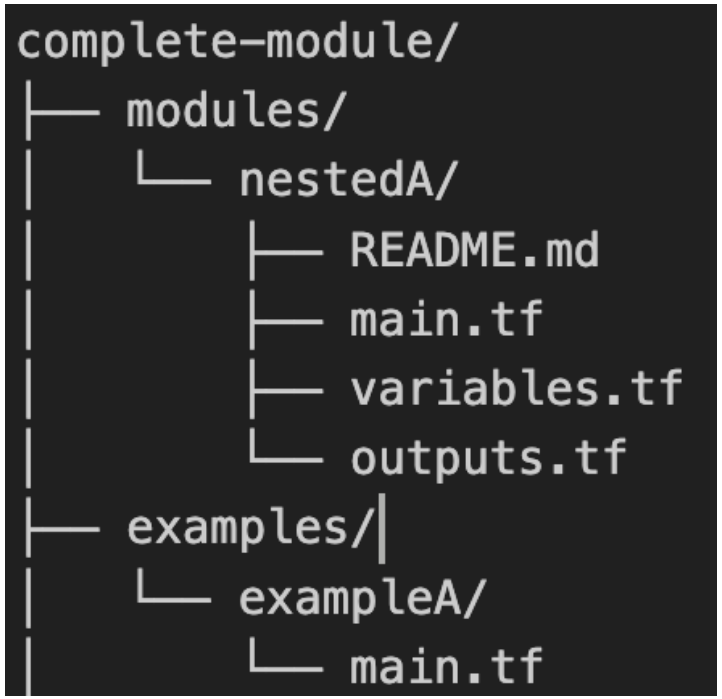
Minimal Recommended Module Layout



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.

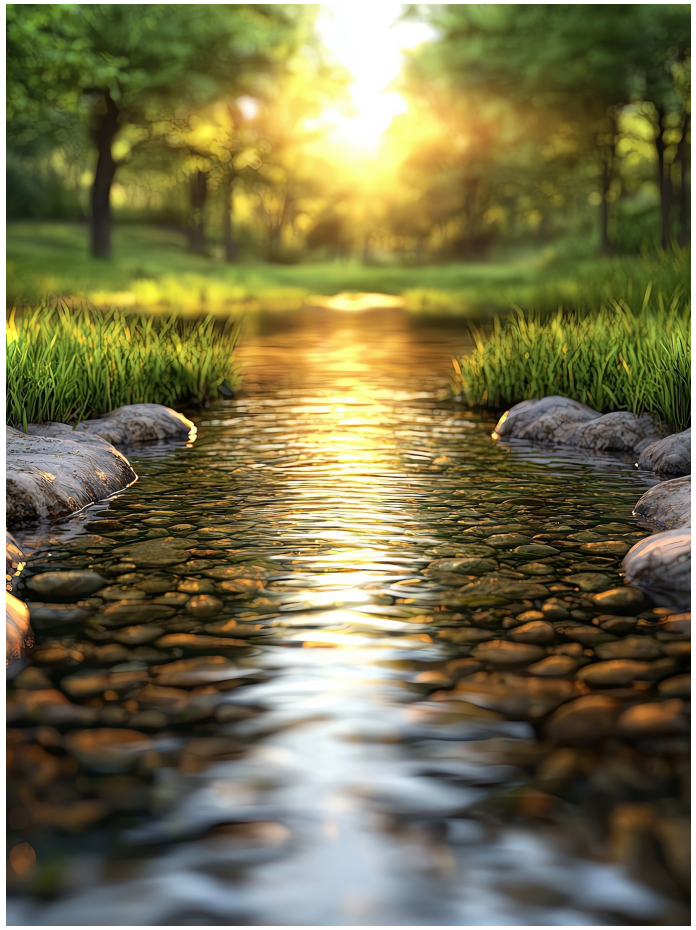
Nested modules are placed in a `modules/` subdirectory, each with its own configuration files and README. Usage examples should be placed in an `examples/` directory, with each example in its own folder. Including a LICENSE file is also recommended for public modules.

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.

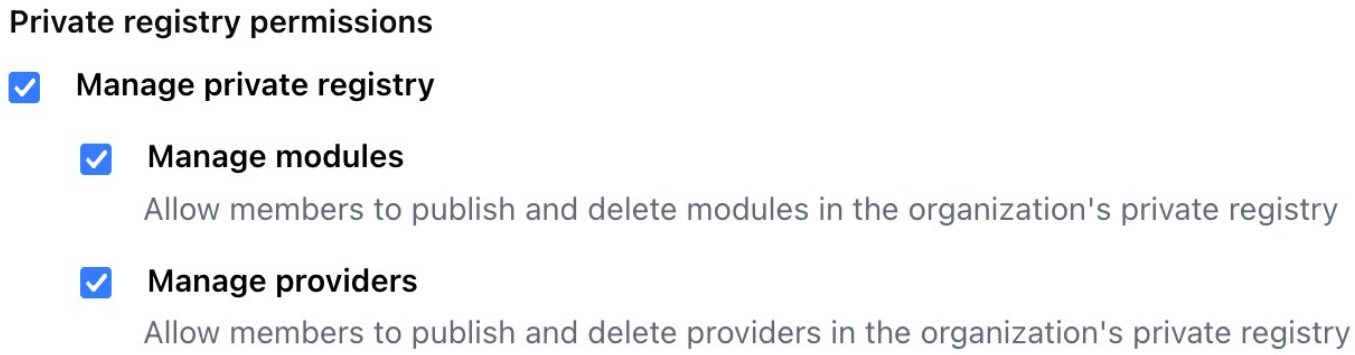
VCS Provider And Repository Requirements

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.

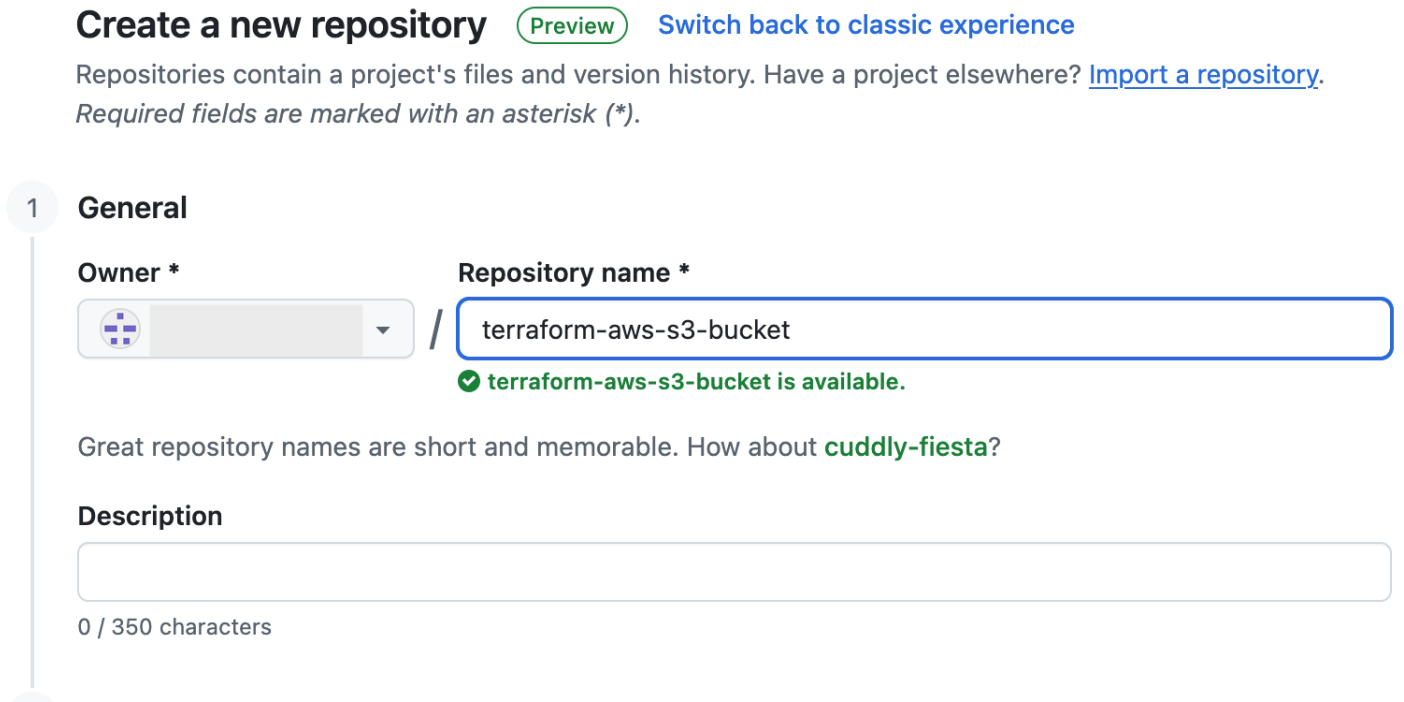
Private Registry Access And Permissions

- 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.

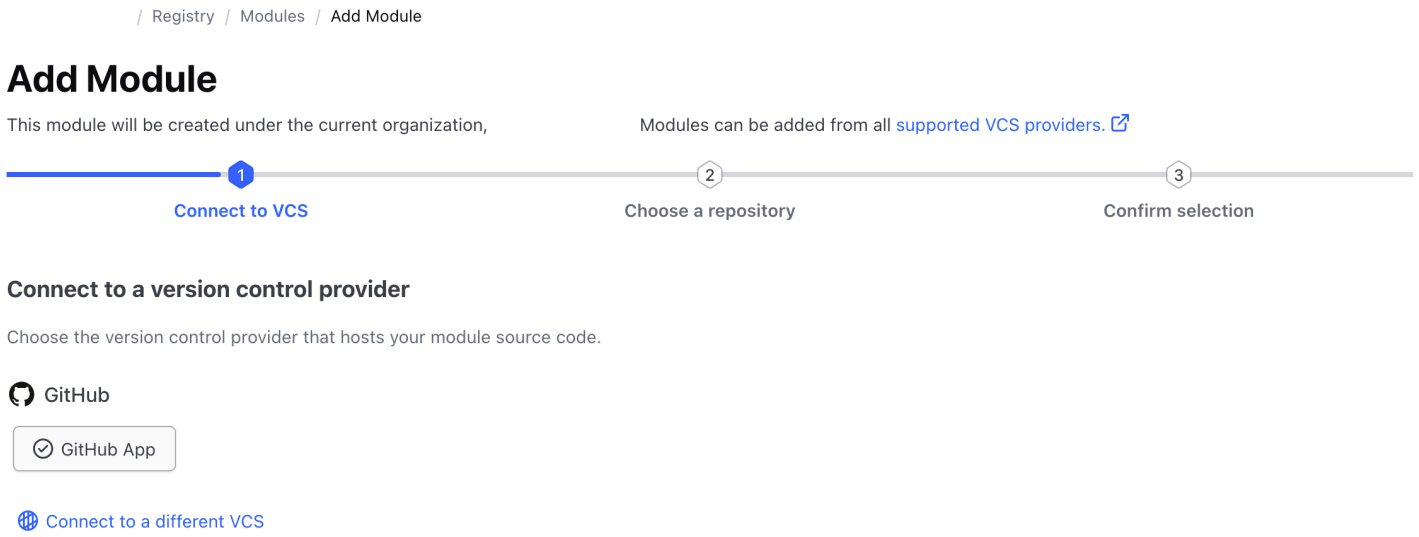
Choose the workflow that best fits your team's release process.

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

Publishing A New Module

To publish a new module:

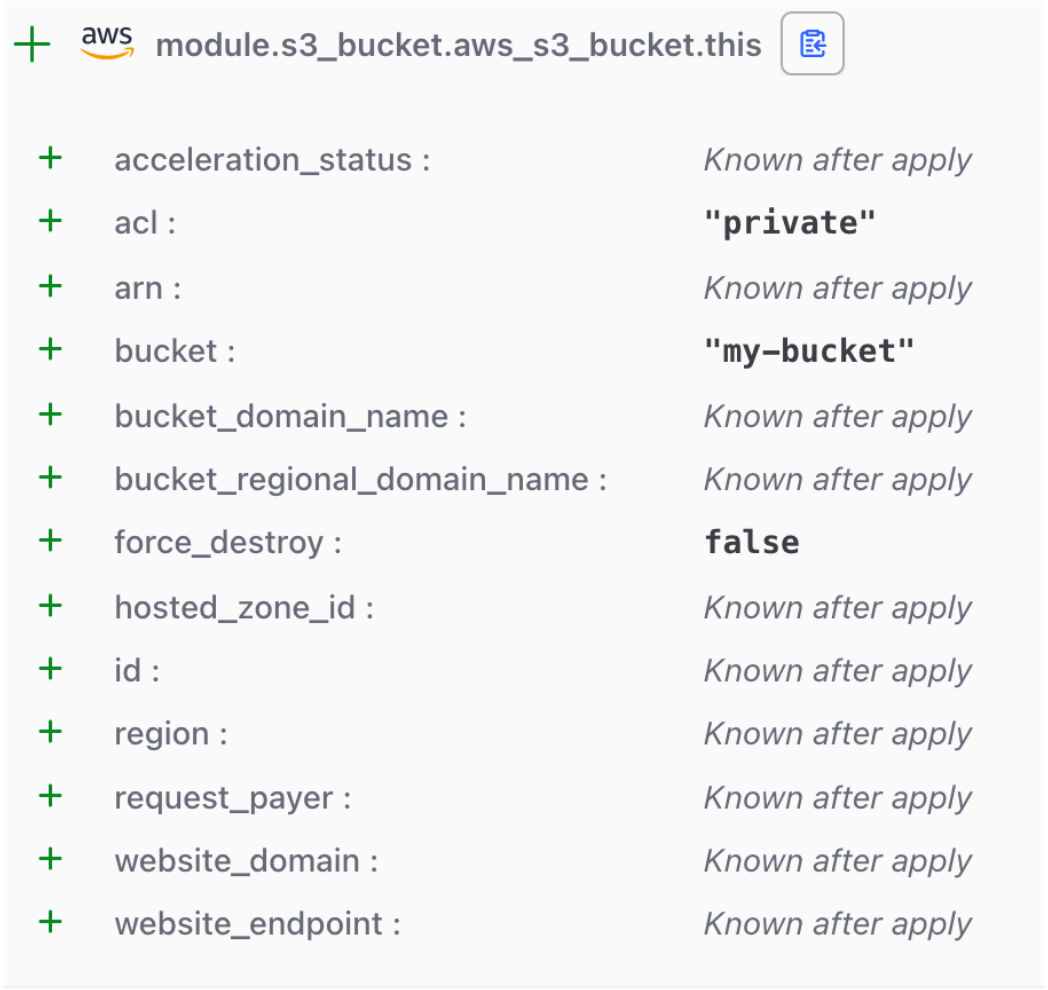
1. Go to the Registry in HCP Terraform and select "Module" from the Publish menu.
2. Choose your VCS connection and repository.
3. Configure the publishing type (tag or branch), specify the source directory, and fill in the module and provider names.
4. Complete the process to add the module to your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

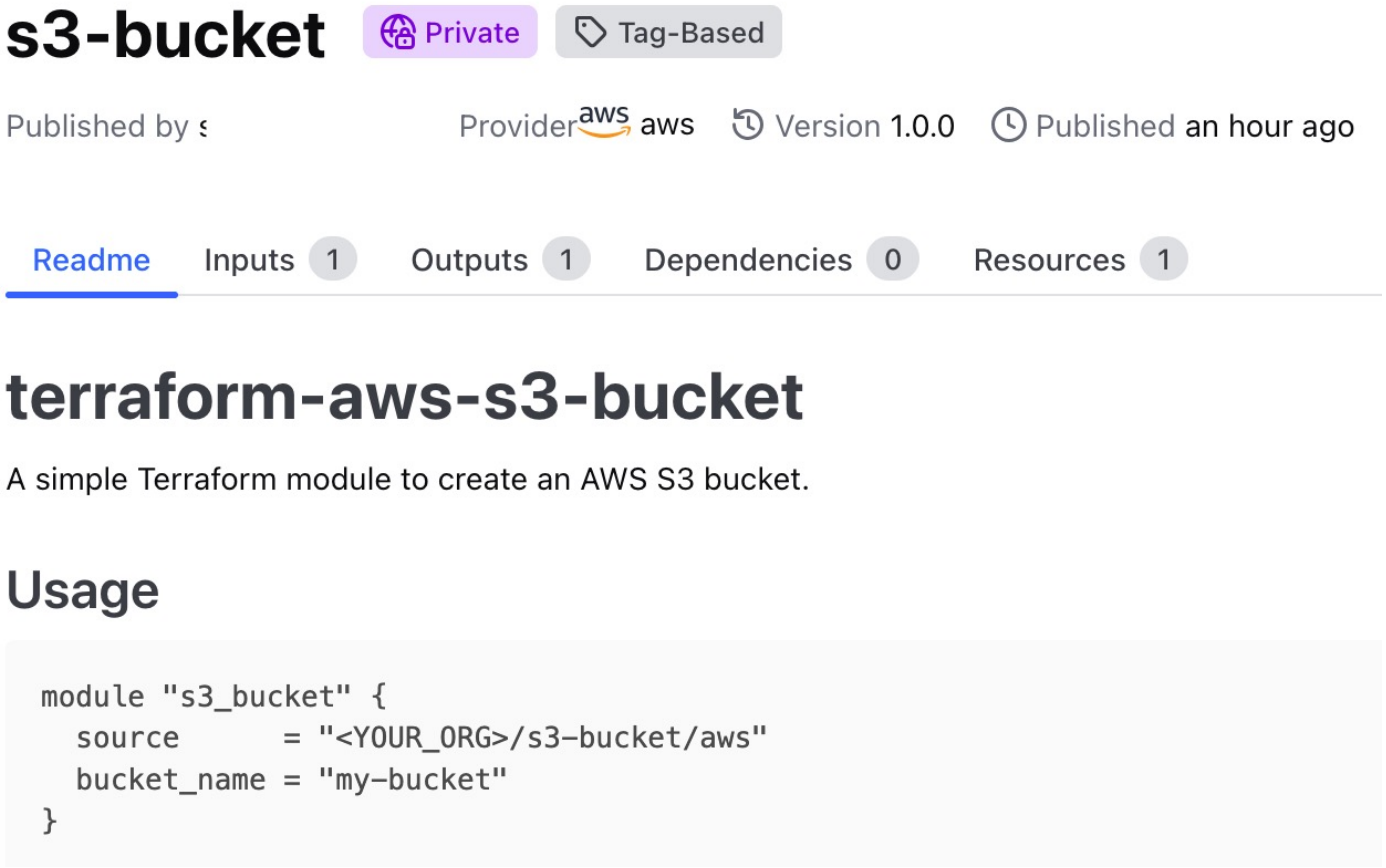
- You can delete individual module versions or entire modules from the registry.
- Deleting a tag in your VCS does not remove the version from the registry; you must delete it in the registry UI.
- If you delete the last version, the module is removed entirely.
- You can delete a module using the API

```
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.

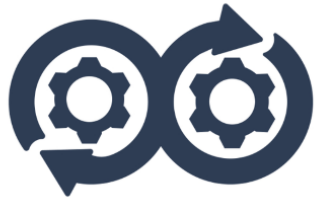


Best Practices And Next Steps

- 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

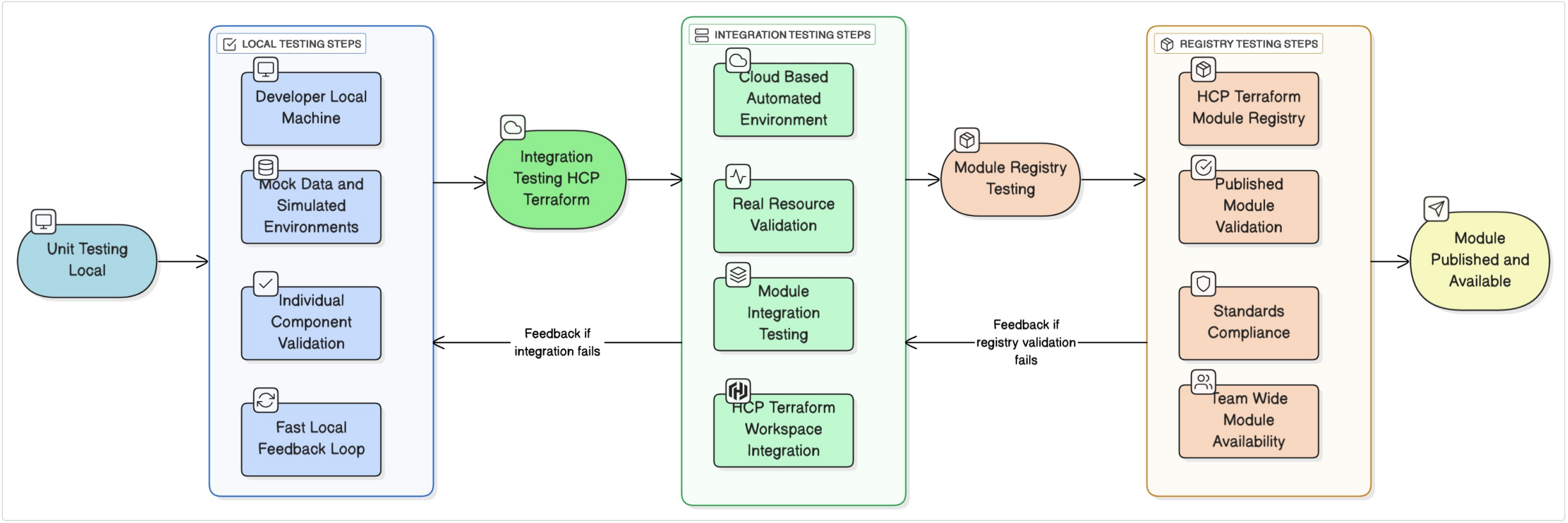


Terraform Testing Framework Overview

Terraform tests allow authors to validate that module configuration updates do not introduce breaking changes. Tests run against test-specific, short-lived resources, preventing any risk to your existing infrastructure or state.

The testing framework is available in Terraform v1.6.0 and later, providing a built-in way to validate your infrastructure as code before it reaches production environments.

Terraform Testing Architecture



Why Test Terraform Configurations?

Testing your Terraform configurations provides several critical advantages for infrastructure management. By implementing a testing strategy, you can catch configuration errors before they affect production infrastructure, ensuring that your deployments remain stable and reliable.

Tests run against isolated, temporary resources without impacting existing systems, giving you confidence to experiment and validate changes. This approach serves as both a safety net and a documentation tool, as tests become living examples of how your modules should behave under various conditions.

How Terraform Testing Works

Terraform tests use .tftest.hcl or .tftest.json files that contain:

- Test blocks configure overall test execution (e.g., parallel vs. sequential)
- Run blocks execute specific Terraform operations (plan/apply) with assertions
- Variables blocks set test-specific input values
- Provider blocks configure test-specific provider settings

Tests can run in two modes: integration testing (creating real infrastructure) or unit testing (plan-only operations).

Testing Approaches

Integration Testing (Default)

- Creates real infrastructure during test execution
- Tests complete Terraform operations
- Validates end-to-end functionality
- Uses command = apply (default)

Unit Testing

- Runs only plan operations without creating resources
- Tests logical operations and custom conditions
- Faster execution and lower cost
- Uses command = plan

Test File Discovery and Components

Terraform automatically identifies test files through their specific file extensions: .tftest.hcl or .tftest.json. These files serve as containers for your infrastructure testing logic.

Every test file is organized into several key sections:

- Test blocks (optional, maximum of one)
- Run blocks (required, can have multiple)
- Variables blocks (optional, maximum of one)
- Provider blocks (optional, can have multiple)

Terraform processes run blocks one after another by default. However, you can configure them to run simultaneously using parallel execution features. Each run block represents a complete Terraform operation within your test configuration directory.

Terraform Test Execution Order

Terraform handles the processing order of variables and provider blocks automatically, regardless of where you place them in your test file. All configuration values are loaded and processed at the start of the test operation.

Following best practices, it's recommended to place your variables and provider blocks at the top of your test file. This organization creates a clear structure and ensures all dependencies are properly initialized before any test execution begins.

Testing AWS EC2 Instance Configuration

The following example demonstrates a basic Terraform configuration that creates an AWS EC2 instance, using an input variable to modify the instance type.

We will create a test file that validates the instance type is configured as expected.

```
# main.tf
provider "aws" {
    region = "us-west-2"
}

variable "instance_type" {
  type = string
}

resource "aws_instance" "web" {
  ami           = "ami-12345678"
  instance_type = var.instance_type
}

output "instance_type" {
  value = aws_instance.web.instance_type
}
```

Test File for Validation

This test file executes a Terraform plan operation to validate the EC2 instance configuration, ensuring the instance type logic works correctly by comparing the configured type against the expected value.

```
# test_instance_type.tftest.hcl
variables {
  instance_type = "t2.micro"
}

run "test_instance_type" {
  command = plan

  assert {
    condition     = aws_instance.web.instance_type == "t2.micro"
    error_message = "Instance type did not match expected"
  }
}
```

Test Block Configuration

The optional test block defines the configuration of the test file, allowing you to configure how the framework executes its runs.

Key Configuration Options:

- parallel: Boolean attribute that enables simultaneous execution of eligible run blocks
- Default value: false (sequential execution)
- When true: Terraform executes all eligible run blocks simultaneously

```
# with_config.tftest.hcl
test {
  parallel = true
}
```

Run Blocks Overview

Run blocks are the core execution units of Terraform tests, simulating Terraform commands within your configuration directory.

Each run block represents a complete Terraform operation and contains the logic for testing your infrastructure configuration.

```
variables {
  instance_type = "t2.micro"
}

run "first" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}
```

Run Block Command and Options

The command attribute and plan_options block tell Terraform which command and options to execute for each run block.

The default operation, if you do not specify a command attribute or the plan_options block, is a normal Terraform apply operation.

Attribute	Description	Default Value
command	Specifies the Terraform operation (plan or apply)	apply
plan_options.mode	Sets planning mode (normal or refresh-only)	normal
plan_options.refresh	Controls state refresh behavior	true
plan_options.replace	List of resources to force replacement	-
plan_options.target	List of resources to target	-
state_key	Controls which state file to use	-
parallel	Enables parallel execution	false

Parallel Execution Configuration

This example demonstrates how parallel execution works in Terraform tests. The test block sets parallel = true globally, but individual run blocks can override this setting.

The first two run blocks execute in parallel, while the third run block sets parallel = false, creating a synchronization point that forces it to wait for all preceding runs to complete.

```
# parallel_execution.tftest.hcl
test {
  parallel = true
}

variables {
  instance_type = "t2.micro"
}

run "first" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}

run "second" {
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}

run "third" {
  parallel = false
  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
}
```

Parallel Execution Configuration II

Terraform can execute run blocks in parallel when the parallel attribute is set to true. Run blocks can execute simultaneously if they don't reference each other's outputs, don't share the same state file, and have parallel=true set.

Parallel execution can be configured globally in the test block or individually in run blocks. A single run block with parallel=false creates a synchronization point, dividing the workflow into groups that must complete sequentially.

```
test {
  parallel = true
}

run "primary_db" {
  state_key = "primary"
}

run "secondary_db" {
  state_key = "secondary"
}

run "site_one" {
  parallel = false  # Creates synchronization point
}
```

Assertions in Terraform Tests

Assertions are the core validation mechanism in Terraform tests, allowing you to verify that your infrastructure configuration behaves as expected. Each assert block contains a condition that must evaluate to true for the test to pass.

Assert blocks have two required attributes:

- condition: A boolean expression that must evaluate to true
- error_message: A descriptive message displayed when the assertion fails

```
run "test_instance_configuration" {
  command = plan

  assert {
    condition     = aws_instance.example.instance_type == "t2.micro"
    error_message = "Instance type should be t2.micro"
  }
```

Variables in Terraform Tests

Variables blocks in Terraform tests allow you to set input values that will be used throughout your test execution. These blocks provide a way to configure your test environment and override default values from your main configuration.

Variables defined in the variables block serve as the default values for your entire test file. However, individual run blocks can override these values using their own variables block, allowing you to test different scenarios with the same configuration.

```
variables {
  bucket_prefix = "test"
  environment   = "development"
  region        = "us-west-2"
}

run "test_bucket_creation" {
  command = plan

  variables {
    bucket_prefix = "production"  # Override for this specific run
  }

  assert {
    condition     = aws_s3_bucket.bucket.bucket == "production-bucket"
    error_message = "Bucket name should use production prefix"
  }
}
```

Variable References in Terraform Tests

Variables in Terraform tests can reference outputs from earlier run blocks and variables defined at higher precedence levels, enabling powerful data flow between test executions.

- Run block variables can reference file-level variables
- Run block variables can reference outputs from previous run blocks
- File-level variables can only reference global variables
- This enables sharing values across multiple run blocks and passing data between different test executions

```
variables {
  global_value = "some value"
}

run "run_block_one" {
  variables {
    local_value = var.global_value
  }

  # Test assertions here
}

run "run_block_two" {
  variables {
    local_value = run.run_block_one.output_one
  }

  # Test assertions here
}
```

Provider Configuration

Providers in Terraform tests can be configured and overridden to control how your infrastructure is deployed during testing. This allows you to customize provider settings specifically for your test environment without affecting your main configuration.

Provider configuration in tests gives you the flexibility to test your infrastructure with different provider configurations, such as using specific regions, credentials, or other provider-specific settings that may differ from your production environment.

```
# customised_provider.tftest.hcl

provider "aws" {
  region = "eu-central-1"
}

variables {
  bucket_prefix = "test"
}

run "valid_string_concat" {
  command = plan

  assert {
    condition     = aws_s3_bucket.bucket.bucket == "test-bucket"
    error_message = "S3 bucket name did not match expected"
  }
}
```

Advanced Provider References in Terraform Tests

Starting from Terraform v1.7.0, provider blocks can reference test file variables and run block outputs, enabling sophisticated provider setup workflows. This capability allows you to dynamically configure providers based on information retrieved from other providers or previous test executions.

This feature enables scenarios where one provider can retrieve credentials or configuration data that is then used to initialize a second provider.

```
provider "vault" {
  # ... vault configuration ...
}

provider "aws" {
  region     = "us-east-1"
  access_key = run.vault_setup.aws_access_key
  secret_key = run.vault_setup.aws_secret_key
}

run "vault_setup" {
  module {
    source = "./testing/vault-setup"
  }
}

run "use_aws_provider" {
  # This run block can use both providers
}
```

Module Configuration

Modules in Terraform tests can be modified to execute different configurations within each run block.

The module block in test files is simplified compared to traditional Terraform modules, supporting only the source and version attributes. Other module configuration options are handled through alternative attributes and blocks within the run block, providing flexibility while maintaining test simplicity.

```
run "test_module_v1" {
  module {
    source  = "./modules/example"
    version = "1.0.0"
  }

  # Other configuration handled in run block
}

run "test_module_v2" {
  module {
    source  = "./modules/example"
    version = "2.0.0"
  }

  # Test different module version
}
```

Advanced Module Usage

When executing alternate modules through the module block, all other run block attributes and blocks remain fully supported. Assert blocks execute against values from the alternate module, enabling comprehensive testing of different module configurations.

Two primary use cases for the modules block include setup modules that create required infrastructure for testing, and loading modules that validate secondary infrastructure like data sources.

```
# instance_count.tftest.hcl

run "setup" {
  # Create the VPC we will use later.

  module {
    source = "./testing/setup"
  }
}

run "execute" {
  # This is empty, we just run the configuration under test using all the default settings.
}

run "verify" {
  # Load and count the instances created in the "execute" run block.

  module {
    source = "./testing/loader"
  }

  assert {
    condition = length(data.aws_instances.instances.ids) == 3
    error_message = "created the wrong number of EC2 instances"
  }
}
```

Module State Management in Terraform Tests

Terraform maintains multiple state files in memory during test execution, with each state file assigned a unique state key for internal tracking. The state key can be overridden using the state_key attribute of a run block to control state file sharing between different modules.

There is always at least one state file for the main configuration under test, shared by all run blocks without alternate modules.

```
run "setup" {
  module {
    source = "./testing/setup"
  }
}

run "init" {
  # Uses main configuration state file
}

run "update_setup" {
  # Reuses setup module state file
  module {
    source = "./testing/setup"
  }
}
```

State Key Override Examples

The state_key attribute allows you to override Terraform's default state file behavior. By default, Terraform creates separate state files for the main configuration and each alternate module, but you can force multiple run blocks to share the same state file regardless of their module source.

This capability is particularly useful when you want to share infrastructure state between run blocks that reference different modules.

```
run "setup" {
  state_key = "main"
  module {
    source = "./testing/setup"
  }
}

run "init" {
  state_key = "main"  # Shares state with setup run
}
```

Module Cleanup in Terraform Tests

Terraform automatically destroys all resources created during test execution when a test file concludes. The destruction order follows reverse run block execution order, which is crucial for maintaining proper resource dependencies during cleanup.

When using alternate modules, Terraform destroys resources in reverse order of when their state files were last referenced. This ensures that dependent resources are destroyed before the resources they depend on.

```
run "setup" {
  # Creates S3 bucket - destroyed LAST
  module {
    source = "./testing/setup"
  }
}

run "execute" {
  # Creates objects in bucket - destroyed SECOND
}

run "verify" {
  # References loader module - destroyed FIRST
  module {
    source = "./testing/loader"
  }
}
```

Expecting Failures in Terraform Tests

Terraform tests support the expect_failures attribute to test failure scenarios. By default, any failed custom conditions cause test failure, but expect_failures allows you to specify which checkable objects should fail for the test to pass.

The expect_failures attribute accepts a list of resources, data sources, check blocks, input variables, and outputs that should fail their custom conditions. This enables testing both positive and negative scenarios within the same test file.

```
variable "input" {
  type = number

  validation {
    condition = var.input % 2 == 1
    error_message = "must be odd number"
  }
}
```

Expecting Failures Implementation

When using expect_failures, be aware that custom conditions (except check blocks) halt Terraform execution. This means you can only reliably include a single checkable object per run block, unless you're only testing check block failures.

Assertions can still be written alongside expect_failures, but they must only reference values computed before the expected failure occurs. Use references or depends_on meta-arguments to manage execution order.

```
variables {
  input = 1
}

run "one" {
  # The variable defined above is odd, so we expect the validation to pass.

  command = plan
}

run "zero" {
  # This time we set the variable is even, so we expect the validation to fail.

  command = plan

  variables {
    input = 0
  }

  expect_failures = [
    var.input,
  ]
}
```

Expecting Failures with Apply Commands

When using expect_failures with command = apply, be aware that the run block will fail if the custom condition fails during the plan phase. This occurs because the apply operation cannot proceed after a failed plan, even when the failure was expected.

While Terraform may not execute some custom conditions during planning (when they depend on computed attributes), it's generally recommended to use expect_failures only with command = plan operations.

```
# This will FAIL because plan fails before apply can run
run "test_apply_failure" {
  command = apply

  variables {
    input = 0  # Even number, will fail validation
  }

  expect_failures = [
    var.input,  # Expects failure but apply never runs
  ]
}

# This works correctly with plan
run "test_plan_failure" {
  command = plan

  variables {
    input = 0  # Even number, will fail validation
  }

  expect_failures = [
    var.input,  # Expects failure and plan shows it
  ]
}
```

Test Mocking

Terraform lets you mock providers, resources, and data sources for your tests. This allows you to test parts of your module without creating infrastructure or requiring credentials. In a Terraform test, a mocked provider or resource will generate fake data for all computed attributes that would normally be provided by the underlying provider APIs.

Mocking functionality can only be used with the terraform test language. Readers of this documentation should be familiar with the testing syntax and language features.

Mock Providers

In Terraform tests, you can mock a provider with the mock_provider block. Mock providers return the same schema as the original provider and you can pass the mocked provider to your tests in place of the matching provider.

All resources and data sources retrieved by a mock provider will set the relevant values from the configuration, and generate fake data for any computed attributes.

```
# bucket_name.tftest.hcl
mock_provider "aws" {}

run "sets_correct_name" {
  variables {
    bucket_name = "my-bucket-name"
  }

  assert {
    condition     = aws_s3_bucket.my_bucket.bucket == "my-bucket-name"
    error_message = "incorrect bucket name"
  }
}
```

Mock Provider Behavior

From the perspective of a plan or apply operation executed in a Terraform test file, the mocked provider is creating actual resources with values that match the configuration.

These resources are stored in the Terraform state files that terraform test creates and holds in memory during test executions.

```
# main.tf
resource "aws_s3_bucket" "my_bucket" {
  bucket = var.bucket_name
}

# test.tftest.hcl
mock_provider "aws" {}

run "test_bucket_creation" {
  variables {
    bucket_name = "test-bucket"
  }

  assert {
    condition     = aws_s3_bucket.my_bucket.bucket == "test-bucket"
    error_message = "Bucket name should match input variable"
  }

  assert {
    condition     = can(aws_s3_bucket.my_bucket.arn)
    error_message = "Bucket should have a generated ARN"
  }
}
```

Mixed Real and Mocked Providers

You can use mocked providers and real providers simultaneously in a Terraform test. The following example defines two AWS providers, one real and one mocked.

You must provide an alias to one of them, as they share the same global aws provider namespace. You can then use the providers attribute in test run blocks to customize which AWS provider to use for each run block.

```
# mocked_providers.tftest.hcl
provider "aws" {}

mock_provider "aws" {
  alias = "fake"
}

run "use_real_provider" {
  providers = {
    aws = aws
  }
}

run "use_mocked_provider" {
  providers = {
    aws = aws.fake
  }
}
```

Mock Provider Data

You can specify specific values for targeted resources and data sources. In a mock_provider block, you can write any number of mock_resource and mock_data blocks.

Both the mock_resource and mock_data blocks accept a type argument that should match the resource or data source you want to provide values for. They also accept a defaults object attribute that you can use to specify the values that should be returned for specific attributes.

```
mock_provider "aws" {
  mock_resource "aws_instance" {
    defaults = {
      arn = "arn:aws:ec2:us-west-1:123456789012:instance/i-12345678"
    }
  }

  mock_data "aws_instance" {
    defaults = {
      arn = "arn:aws:ec2:us-west-1:123456789012:instance/i-12345678"
    }
  }
}
```

Mock Provider Overrides

In addition to mocking providers, you can use the following block types to override specific resources, data sources, and modules:

• override_resource: Override the values of a resource. Terraform does not call the underlying provider.

• override_data: Override the values of a data source. Terraform does not call the underlying provider.

• override_module: Override the outputs of a module. Terraform does not create any resource in the module.

```
mock_provider "aws" {}

override_data {
  target = data.aws_availability_zones.available
  values = {
    names = ["us-west-1a", "us-west-1b", "us-west-1c"]
  }
}
```

Lab: Terraform Unit Testing