Terraform Certification

Jacopo Pela

Luglio 2023

Contents

Terraform is an immutable, declarative, Infrastructure as Code provisioning language based on Hashicorp Configuration Language, or optionally JSON.

When using the CLI-driven workflow for Terraform Cloud, any variables passed using the -var flag will override workspace-specific variables

Terraform is a tool that lets you define infrastructure in human and machine-readable code.
HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share.

1 How Terraform Work

Terraform creates and manages resources on cloud platforms and other services through their application programming interfaces (APIs).
Providers enable Terraform to work with virtually any platform or service with an accessible API.

The core Terraform workflow consists of three stages:

- **Write**: You define resources, which may be across multiple cloud providers and services.
- **Plan**: Terraform creates an execution plan describing the infrastructure it will create, update, or destroy based on the existing infrastructure and your configuration.
- **Apply**: On approval, Terraform performs the proposed operations in the correct order, respecting any resource dependencies.

Terraform generates a plan and prompts you for your approval before modifying your infrastructure. It also keeps track of your real infrastructure in a state file, which acts as a source of truth for your environment.
The state is saved under the **terraform.tfstate** file

1.1 What is IAC

Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface.
Terraform builds a resource graph to determine resource dependencies and creates or modifies non-dependent resources in parallel.

Providers define individual units of infrastructure, for example compute instances or private networks, as resources. You can compose resources from different providers into reusable Terraform configurations called modules, and manage them with a consistent language and workflow.

Terraform's configuration language is declarative, meaning that it describes the desired end-state for your infrastructure.
Terraform providers automatically calculate dependencies between resources to create or destroy them in the correct order.

To deploy infrastructure with Terraform:

- Scope - Identify the infrastructure for your project.
- Author - Write the configuration for your infrastructure.
- Initialize - Install the plugins Terraform needs to manage the infrastructure.
- Plan - Preview the changes Terraform will make to match your configuration.
- Apply - Make the planned changes.

1.2 Build Infrastructure

1.2.1 Terraform Block

The terraform block contains Terraform settings, including the required providers Terraform will use to provision your infrastructure.
For each provider, the source attribute defines an optional hostname, a namespace, and the provider type. Terraform installs providers from the Terraform Registry by default.

You can also set a version constraint for each provider defined in the **required_providers** block. The **version** attribute is optional, but we recommend using it to constrain the provider version so that Terraform does not install a version of the provider that does not work with your configuration. If you do not specify a provider version, Terraform will automatically download the most recent version during initialization.

1.2.2 Providers

The **provider** block configures the specified provider. A provider is a plugin that Terraform uses to create and manage your resources.
You can use multiple provider blocks in your Terraform configuration to manage resources from different providers.

To select a provider alias use the following syntax inside a resource block: **provider = <provider_name>.<alias>**

To update the provider lock use the following command: **terraform providers lock**

To specify a required version use **required_providers** variable inside a provider block

1.2.3 Resources

Use **resource** blocks to define components of your infrastructure. A resource might be a physical or virtual component, or it can be a logical resource.
Resource blocks have two strings before the block: the resource type and the resource name.
Together, the resource type and resource name form a unique ID for the resource.
Resource blocks contain arguments which you use to configure the resource.

1.3 Initialize the directory

When you create a new configuration you need to initialize the directory with **terraform init**.

Initializing a configuration directory downloads and installs the providers defined in the configuration.
The **terraform init** command prints out which version of the provider was installed.
Terraform also creates a lock file named **.terraform.lock.hcl** which specifies the exact provider versions used, so that you can control when you want to update the providers used for your project.

1.4 Format and validate the configuration

We recommend using consistent formatting in all of your configuration files. The **terraform fmt** command automatically updates configurations in the current directory for readability and consistency.
You can also make sure your configuration is syntactically valid and internally consistent by using the **terraform validate** command.

1.5 Create infrastructure

Apply the configuration now with the **terraform apply** command.
Before it applies any changes, Terraform prints out the execution plan which describes the actions Terraform will take in order to change your infrastructure to match the configuration.
Terraform will now pause and wait for your approval before proceeding. If anything in the plan seems incorrect or dangerous, it is safe to abort here before Terraform modifies your infrastructure.

To rebuild a single resource you can use the **terraform taint**

1.6 Inspect State

When you applied your configuration, Terraform wrote data into a file called **terraform.tfstate**. Terraform stores the IDs and properties of the resources it manages in this file, so that it can update or destroy those resources going forward.

The Terraform state file is the only way Terraform can track which resources it manages, and often contains sensitive information, so you must store your state file securely and restrict access to only trusted team members who need to manage your infrastructure.

1.7 Manually Managing State

Terraform has a built-in command called **terraform state** for advanced state management.
To see the list of the resources deloyed and stored int the state file use the **list** flag.
To see the resource block of a specific resource use the commang **terraform state show** followed by the resource name specified with the following syntax:
<resorce_type>.<resource_name>

1.8 Destroy Infrastructure

The terraform destroy command terminates resources managed by your Terraform project. This command is the inverse of terraform apply in that it terminates all the resources specified in your Terraform state. It does not destroy resources running elsewhere that are not managed by the current Terraform project.
Just like with apply, Terraform determines the order to destroy your resources. In this case, Terraform identified a single instance with no other dependencies, so it destroyed the instance.

1.9 Define a variable

The current configuration includes a number of hard-coded values.
Terraform variables allow you to write configuration that is flexible and easier to re-use.

```
variable "instance_name" {
  description = "Value of the Name tag for the EC2 instance"
  type        = string
  default     = "ExampleAppServerInstance"
}
```

The variable block will default to its default value unless you declare a different value.
Overriding the default instance name by passing in a variable using the **-var** flag.

1.10 Query Data with Outputs

Add the configuration below to define outputs for your EC2 instance's ID and IP address.

```
output "instance_id" {
  description = "ID of the EC2 instance"
  value       = aws_instance.app_server.id
}
```

Terraform prints output values to the screen when you apply your configuration.
Query the outputs with the **terraform output** command.

1.11 Store Remote State

In production environments you should keep your state secure and encrypted, where your teammates can access it to collaborate on infrastructure.
The best way to do this is by running Terraform in a remote environment with shared access to state.

2 Purpose of Terraform State

State is a necessary requirement for Terraform to function.

Terraform requires some sort of database to map Terraform config to the real world.
When you have a resource **resource "aws_instance" "foo"** in your configuration, Terraform uses this map to know that instance **i-abcd1234** is represented by that resource.
For mapping configuration to resources in the real world, Terraform uses its own state structure.
Terraform expects that each remote object is bound to only one resource instance.

2.1 Metadata

Alongside the mappings between resources and remote objects, Terraform must also track metadata such as resource dependencies.
Terraform typically uses the configuration to determine dependency order.
To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state.
Terraform also stores other metadata for similar reasons, such as a pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

2.2 Performance

In addition to basic mapping, Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.

When running a **terraform plan**, Terraform must know the current state of resources in order to effectively determine the changes that it needs to make to reach your desired configuration.
For larger infrastructures, querying every resource is too slow.

2.3 Syncing

Terraform stores the state in a file in the current working directory where Terraform was run.
When using Terraform in a team it is important for everyone to be working with the same state so that operations will be applied to the same remote objects.
Remote state is the recommended solution to this problem.

3 Manage Terraform versions

Use the **required_version** setting to control when you upgrade the version of Terraform that you use for your Terraform projects to make updates more

predictable.
Configuration sets **required_version** with **~>0.12.29**.
The ~> symbol allows the patch version to be greater than 29 but requires the major and minor versions (0.12) to match the version that the configuration specifies

Use the **version** subcommand to check your Terraform version and the version of any providers your configuration is using.

New minor and patch versions of Terraform are backward compatible with configuration written for previous versions.

When you run Terraform commands, Terraform stores its current version in your project's state file,

Once you use a newer version of Terraform's state file format on a given project, there is no supported way to revert to using an older state file version.

3.1 Providers

Terraform relies on plugins called "providers" to interact with cloud providers, SaaS providers, and other APIs.

Terraform configurations must declare which providers they require so that Terraform can install and use them

3.1.1 What Providers Do

Each provider adds a set of resource types and/or data sources that Terraform can manage.

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

3.1.2 How to Use Providers

To use resources from a given provider, you need to include some information about it in your configuration.
Terraform Cloud and Terraform Enterprise install providers as part of every run
Terraform CLI finds and installs providers when initializing a working directory. It can automatically download providers from a Terraform registry, or load them from a local mirror or cache.

3.2 How Terraform Works With Plugins

Terraform is built on a plugin-based architecture, enabling developers to extend Terraform by writing new plugins or compiling modified versions of existing plugins.

3.2.1 Terraform Plugins

Terraform Plugins are written in Go and are executable binaries invoked by Terraform Core over RPC.
All Providers and Provisioners used in Terraform configurations are plugins. They are executed as a separate process and communicate with the main Terraform binary over an RPC interface.

When terraform init is run with the **-upgrade option**, it re-checks the Terraform Registry for newer acceptable provider versions and downloads them if available.

3.3 Dependency Lock File

TODO

The dependency lock file is a file that belongs to the configuration as a whole, rather than to each separate module in the configuration. For that reason Terraform creates it and expects to find it in your current working directory when you run Terraform.

The lock file is always named **.terraform.lock.hcl**

Terraform automatically creates or updates the dependency lock file each time you run the terraform init command.
The dependency lock file uses the same low-level syntax as the main Terraform language.

4 Navigate the core workflow

4.1 The Core Terraform Workflow

The core Terraform workflow has three steps:

- **Write** - Author infrastructure as code.
- **Plan** - Preview changes before applying.
- **Apply** - Provision reproducible infrastructure.

4.1.1 Write

You write Terraform configuration just like you write code: in your editor of choice
HashiCorp Terraform recommends to write 2 spaces between each nesting level
As you make progress on authoring your config, repeatedly running plans can help flush out syntax errors and ensure that your config is coming together as you expect.

4.1.2 Plan

When the feedback loop of the **Write** step has yielded a change that looks good, it's time to commit your work and review the final plan.

4.1.3 Apply

After one last check, you are ready to tell Terraform to provision real infrastructure.

This core workflow is a loop; the next time you want to make changes, you start the process over from the beginning.

4.2 Command: init

The **terraform init** command is used to initialize a working directory containing Terraform configuration files.
This is the first command that should be run after writing a new Terraform configuration.
It is safe to run this command multiple times.

4.2.1 Usage

This command performs several different initialization steps in order to prepare the current working directory for use with Terraform.
This command will never delete your existing configuration or state.

4.2.2 Copy a Source Module

By default, **terraform init** assumes that the working directory already contains a configuration and will attempt to initialize that configuration.
Optionally, init can be run against an empty directory with the **-from-module=MODULE-SOURCE** option, in which case the given module will be copied into the target directory before any other initialization steps are run.

4.2.3 Backend Initialization

During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.
Re-running init with an already-initialized backend will update the working directory to use the new backend settings.
Either **-reconfigure** or **-migrate-state** must be supplied to update the backend configuration.
The **-migrate-state** option will attempt to copy existing state to the new backend.
The **-reconfigure** option disregards any existing configuration, preventing

migration of any existing state.

4.2.4 Child Module Installation

During init, the configuration is searched for **module** blocks, and the source code for referenced modules is retrieved from the locations given in their source arguments.
Re-running init with modules already installed will install the sources for any modules that were added to configuration since the last init, but will not change any already-installed modules. Use **-upgrade** to override this behavior, updating all modules to the latest available source code.

4.2.5 Plugin Installation

Most Terraform providers are published separately from Terraform as plugins. During init, Terraform searches the configuration for both direct and indirect references to providers and attempts to install the plugins for those providers.
For providers that are published in either the public Terraform Registry or in a third-party provider registry, **terraform** init will automatically find, download, and install the necessary provider plugins.
After successful installation, Terraform writes information about the selected providers to the dependency lock file.

4.3 Command: validate

The **terraform validate** command validates the configuration files in a directory.
Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state.
Validation requires an initialized working directory with any referenced plugins and modules installed.

To output the result of the **terraform validate** command in JSON format use the **-json** flag.

4.4 Command: plan

The **terraform plan** command creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure.

when Terraform creates a plan it:

- Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
- Compares the current configuration to the prior state and noting any differences.
- Proposes a set of change actions that should, if applied, make the remote objects match the configuration.

The plan command alone will not actually carry out the proposed changes, and so you can use this command to check whether the proposed changes match what you expected.
You can use the optional **-out=FILE** option to save the generated plan to a file on disk, which you can later execute by passing the file to **terraform apply**.

4.4.1 Planning Modes

Terraform has two alternative planning modes, each of which creates a plan with a different intended outcome.
Destroy mode: creates a plan whose goal is to destroy all remote objects that currently exist, leaving an empty Terraform state.

Activate destroy mode using the **-destroy** command line option.

Refresh-only mode: creates a plan whose goal is only to update the Terraform state and any root module output values to match changes made to remote objects outside of Terraform.

4.4.2 Resource Targeting

You can use the **-target** option to focus Terraform's attention on only a subset of resources. You can use resource address syntax to specify the con-

straint.
Once Terraform has selected one or more resource instances that you've directly targeted, it will also then extend the selection to include all other objects that those selections depend on either directly or indirectly.

4.5 Command: apply

The **terraform apply** command executes the actions proposed in a Terraform plan.
Another way to use **terraform apply** is to pass it the filename of a saved plan file you created earlier with **terraform plan -out=...**.
This two-step workflow is primarily intended for when running Terraform in automation.

4.5.1 Usage

The behavior of **terraform apply** differs significantly depending on whether you pass it the filename of a previously-saved plan file.

4.5.2 Saved Plan Mode

If you pass the filename of a previously-saved plan file, **terraform apply** performs exactly the steps specified by that plan file. It does not prompt for approval

4.5.3 Plan Options

When run without a saved plan file, **terraform apply** supports all of **terraform plan**'s planning modes and planning options.

4.5.4 Passing a Different Configuration Directory

If your workflow relies on overriding the root module directory, use the **-chdir** global option, which works across all commands and makes Terraform consistently look in the given directory for all files it would normally read or write in the current working directory.

Terraform does not support rolling back a partially-completed apply. Because of this, your infrastructure may be in an invalid state after a Terraform

apply step errors out. After you resolve the error, you must apply your configuration again to update your infrastructure to the desired state.

The **terraform show** command prints out Terraform's current understanding of the state of your resources. It does not refresh your state, so the information in your state can be out of date

4.5.5 Replace Resources

When using Terraform, you will usually apply an entire configuration change at once. Terraform and its providers will determine the changes to make and the order to make them in.
There are some cases where you may need to replace or modify individual resources. Terraform provides two arguments to the apply command that allow you to interact with specific resources: **-replace** and **-target**.

Use the **-replace** argument when a resource has become unhealthy or stops working in ways that are outside of Terraform's control.
The **-replace** argument requires a resource address. The second case where you may need to partially apply configuration is when troubleshooting an error that prevents Terraform from applying your entire configuration at once.

4.6 Command: destroy

The **terraform destroy** command is a convenient way to destroy all remote objects managed by a particular Terraform configuration.

4.6.1 Usage

This command is just a convenience alias for the following command:

```
terraform apply -destroy
```

For that reason, this command accepts most of the options that **terraform apply** accepts

4.7 Command: fmt

The **terraform fmt** command is used to rewrite Terraform configuration files to a canonical format and style.
Other Terraform commands that generate Terraform configuration will produce configuration files that conform to the style imposed by terraform fmt, so using this style in your own files will ensure consistency.

To see the difference introduced with the **terrafrom fmt** command without impact the current code use the **-diff** flag.

To call the **fmt** commad recursively use the **-recursive** flag.

Formatting decisions are always subjective and so you might disagree with the decisions that **terraform fmt** makes. This command is intentionally opinionated and has no customization options because its primary goal is to encourage consistency of style between different Terraform codebases.

5 Learn more subcommands

The Terraform CLI includes subcommands for operations beyond the core workflow, including importing resources and manipulating and inspecting state

5.1 State Command

The **terraform state** command is used for advanced state management.
Rather than modify the state directly, the **terraform state** commands can be used in many cases

5.1.1 Remote State

The Terraform state subcommands all work with remote state just as if it was local state. Reads and writes may take longer than normal as each read and each write do a full network roundtrip. Otherwise, backups are still written to disk and the CLI usage is the same as if it were local state.

5.1.2 Backups

All **terraform state** subcommands that modify the state write backup files. The path of these backup file can be controlled with **-backup**.
Subcommands that are read-only do not write any backup files since they aren't modifying the state.
Backups for state modification can not be disabled. Due to the sensitivity of the state file, Terraform forces every state modification command to write a backup file.

5.2 Manage resources in Terraform state

Terraform stores information about your infrastructure in a state file. This state file keeps track of resources created by your configuration and maps them to real-world resources.
Terraform compares your configuration with the state file and your existing infrastructure to create plans and make changes to your infrastructure.

You should not manually change information in your state file in a real-world situation to avoid unnecessary drift between your Terraform configuration, state, and infrastructure.

Because your state file has a record of your dependencies, enforced by you with a **depends_on** attribute or by Terraform automatically, any changes to the dependencies will force a change to the dependent resource.

5.2.1 Examine State with CLI

The Terraform CLI allows you to review resources in the state file without interacting with the **.tfstate** file itself.
Run **terraform show** to get a human-friendly output of the resources contained in your state.
Run **terraform state list** to get the list of resource names and local identifiers in your state file

5.2.2 Replace a resource with CLI

Terraform usually only updates your infrastructure if it does not match your configuration.

You can use the **-replace** flag for **terraform plan** and **terraform apply** operations to safely recreate resources in your environment even if you have not edited the configuration.
The **-replace** flag allows you to target specific resources and avoid destroying all the resources in your workspace just to fix one of them.

5.2.3 Remove a resource from state

The **terraform state rm** subcommand removes specific resources from your state file.
This does not remove the resource from your configuration or destroy the infrastructure itself.

5.2.4 Refresh modified infrastructure

The **terraform refresh** command updates the state file when physical resources change outside of the Terraform workflow.
refresh command don't process change in the configuration file.
If you are running a terraform version older than v0.15.4 you can use the **refresh-only** flag insted of the **refresh** command

Terraform automatically performs a refresh during the plan, apply, and destroy operations. All of these commands will reconcile state by default, and have the potential to modify your state file.

5.3 Command: import

The **terraform import** command is used to import existing resources into Terraform.

5.4 Usage

Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS.

5.4.1 Provider Configuration

Terraform will attempt to load configuration files that configure the provider being used for import. If no configuration files are present or no configuration

for that specific provider is present, Terraform will prompt you for access credentials.

5.5 Import Usage

The **terraform import** command can only import one resource at a time. It cannot simultaneously import an entire collection of resources

5.5.1 Complex Imports

An import may also result in a "complex import" where multiple resources are imported

5.6 Debugging Terraform

Terraform has detailed logs which can be enabled by setting the **TF_LOG** environment variable to any value.
You can set **TF_LOG** to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.

Logging can be enabled separately for terraform itself and the provider plugins using the **TF_LOG_CORE** or **TF_LOG_PROVIDER** environment variables.

To persist logged output you can set **TF_LOG_PATH** in order to force the log to always be appended to a specific file when logging is enabled.

6 Use and create modules

Modules help you organize and re-use Terraform configuration

6.1 Modules overview

In many ways, Terraform modules are similar to the concepts of libraries, packages, or modules found in most programming languages, and provide many of the same benefits.

6.1.1 What is a Terraform module?

A Terraform module is a set of Terraform configuration files in a single directory.
When you run Terraform commands directly from such a directory, it is considered the **root module**

6.1.2 Calling modules

Terraform commands will only directly use the configuration files in one directory, which is usually the current working directory. Your configuration can use module blocks to call modules in other directories.
A module that is called by another configuration is sometimes referred to as a "child module" of that configuration.

6.1.3 Local and remote modules

Modules can either be loaded from the local filesystem, or a remote source. Terraform supports a variety of remote sources

6.2 Use registry modules in configuration

6.3 Set values for module input variables

Modules can contain both required and optional arguments. You must specify all required arguments to use the module. Most module arguments correspond to the module's input variables. Optional inputs will use the module's default values if not defined.

6.3.1 Review root input variables

Using input variables with modules is similar to using variables in any Terraform configuration.
You can pass the variables to the module block as arguments.
You do not need to set all module input variables with variables.

6.3.2 Review root output variables

Modules also have output values.
You can reference module outputs in other parts of your configuration. Ter-

raform will not display module outputs by default. You must create a corresponding output in your root module and set it to the module's output

6.3.3 Understand how modules work

When using a new module for the first time, you must run either **terraform init** or **terraform get** to install the module. When you run these commands, Terraform will install any new modules in the **.terraform/modules** directory within your configuration's working directory.

When Terraform processes a module block, it will inherit the provider from the enclosing configuration Variables within modules work almost exactly the same way that they do for the root module.
When using a module, variables are set by passing arguments to the module in your configuration.
Variables declared in modules that aren't given a default value are required, and so must be set whenever you use the module.

You should also consider which values to add as outputs, since outputs are the only supported way for users to get information about resources configured by the module.

You can access a module's output from the configuration that calls the module through the following syntax: **module.<MODULE NAME>.<OUTPUT NAME>.**

6.3.4 Install the local module

Whenever you add a new module to a configuration, Terraform must install the module before it can be used. Both the **terraform get** and **terraform init** commands will install and update modules. The **terraform init** command will also initialize backends and install plugins.

6.4 Customize modules with object attributes

Object type attributes contain a fixed set of named values of different types. Using objects in your modules lets you group related attributes together, making it easier for users to understand how to use your module

6.4.1 Refactor module with object attribute

Objects map a specific set of named keys to values. Keeping related attributes in a single object helps your users understand how to use your module.

6.5 Share modules in the private registry

Terraform Cloud allows users to create and confidentially share infrastructure modules within an organization using the private registry.

In order to publish modules to the Terraform registry, module names must have the format **terraform-<PROVIDER>-<NAME>**, where **<NAME>** can contain extra hyphens.

Terraform Cloud modules should be semantically versioned, and pull their versioning information from repository release tags. To publish a module initially, at least one release tag must be present.

6.5.1 Create a configuration that uses the module

Modules from the private registry can be referenced using a registry source address of the form:
app.terraform.io/<ORGANIZATION-NAME>/terraform/<NAME>/<PROVIDER>.

6.6 Add public providers and modules to your private registry

Curating public modules and providers in your private registry lets you define a list of approved components for your organization to use. It also lets your team find all documentation related to those components in one place.

6.7 Refactor monolithic Terraform configuration

Defining multiple environments in the same main.tf file may become hard to manage as you add more resources.
Terraform loads all configuration files within a directory and appends them

together, so any resources or providers with the same name in the same directory will cause a validation error.
When working with monolithic configuration, you can use the **terraform apply** command with the **-target** flag to scope the resources to operate on, but that approach can be risky and is not a sustainable way to manage distinct environments.
For safer operations, you need to separate your state.
There are two primary methods to separate state between environments: directories and workspaces.

6.8 Module creation - recommended pattern

Terraform modules are self-contained pieces of infrastructure-as-code that abstract the underlying complexity of infrastructure deployments.

6.8.1 Module creation workflow

Modules should be opinionated and designed to do one thing well.
If a module's function or purpose is hard to explain, the module is probably too complex.
When initially scoping your module, aim for small and simple to start.

- **Encapsulation**: Group infrastructure that is always deployed together.
- **Privileges**: Restrict modules to privilege boundaries.
- **Volatility**: Separate long-lived infrastructure from short-lived.

6.8.2 Create the module MVP

Modules, like any piece of code, are never complete.
There will always be new module requirements and changes.
Embrace this and aim for your first few module versions to meet the minimum viable product (MVP) standards.
Output as much information as possible from your module MVP even if you do not currently have a use for it.

6.8.3 Nesting modules

A nested module is a reference to invoke another module from the current module.
Nested modules can be located externally and are referred to as "child modules", or embedded inside the current workspace and are referred to as "submodules".
Nesting modules can speed development, but can lead to unclear and unexpected outcomes.

External modules are often centrally managed and versioned so that new releases are validated before consumers can use them.
A change to the nested module can affect the parent module with no changes to the parent's calling code or version.

Embedding one or more submodule inside your current code base enables you to cleanly separate logical components of the primary module or to create a reusable code block that can be invoked multiple times during the execution of the calling module.

6.8.4 Label and document module elements

Create and follow a naming convention for your module elements to make them easier to understand and work with.

6.9 Use configuration to move resources

As your Terraform configuration grows in complexity, updating resources becomes more risky: an update to one resource may cause unintended changes to other parts of your infrastructure. One way to address this is to refactor your existing Terraform code into separate modules.

When you move existing resources from a parent to a child module, your Terraform resource IDs will change. Because of this, you must let Terraform know that you intend to move resources rather than replace them, or Terraform will destroy and recreate your resources with the new ID.

The **moved** configuration block lets you track your resource moves in the configuration itself.

6.9.1 Rename and move a resource

You can also use the **moved** configuration block to rename existing resources.

We strongly recommend you retain all **moved** blocks in your configuration as a record of your changes. Removing a **moved** block plans to delete that existing resource instead of moving it.

6.10 Create and use no-code modules

No-code provisioning in Terraform Cloud lets users deploy infrastructure resources without writing Terraform configuration.
Modules can codify your infrastructure standards and architecture requirements, making it easier for Terraform configuration authors to deploy infrastructure that complies with best practices.
No-code provisioning lets users deploy infrastructure in modules without writing any Terraform configuration

No-code modules are available in Terraform Cloud Plus Edition.

6.10.1 Review module design recommendations

Unlike standard module deployment, users do not provision infrastructure in no-code modules by referencing them in written configuration. Because of this, you must write no-code modules in a specific way.

No-code modules must follow standard module structure and define all resources in the root repository of the directory.
The main difference between no-code modules and ordinary modules is that the no-code workflow requires declaring provider configuration within the module itself.

When users provision infrastructure with a no-code module, Terraform Cloud will automatically launch a new workspace to manage the module's resources. Because no-code modules contain their provider configuration, organization administrators must also enable automatic access to provider credentials.

6.10.2 Limit configurable variables and attributes

reduce the number of decisions the user needs to make.
A well-designed no-code module is scoped to a specific use case and limits the number of variables a user needs to configure.

The no-code provisioning workflow prompts users to set values for the module's input variables that do not have defaults before creating the new workspace and deploying resources.
The new workspace will also access any global variable sets in your organization

6.10.3 Publish no-code ready module

Module repositories published to the Terraform registry must follow the name format **terraform-<provider>-<name>** and have semantically versioned tags associated with releases.

When you enable no-code provisioning on a module, Terraform Cloud displays a **No-Code Ready** badge next to the module name and adds a **Provision Workspace** button to the details page.

6.11 Finding and Using Modules

6.11.1 Finding Modules

By default, only verified modules are shown in search results. Verified modules are reviewed by HashiCorp to ensure stability and compatibility.

6.11.2 Using Modules

The Terraform Registry is integrated directly into Terraform, so a Terraform configuration can refer to any module published in the registry.
The syntax for specifying a registry module is:
<NAMESPACE>/<NAME>/<PROVIDER>

Module registry integration was added in Terraform v0.10.6, and full versioning support in v0.11.0

Some modules have required inputs you must set before being able to use the module.

The **terraform init** command will download and cache any modules referenced by a configuration.

6.11.3 Private Registry Module Sources

You can also use modules from a private registry.
Private registry modules have source strings of the form:
<HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>.
This is the same format as the public registry, but with an added hostname prefix. eg:

```
module "vpc" {
  source = "app.terraform.io/example_corp/vpc/aws"
  version = "0.9.3"
}
```

Depending on the registry you're using, you might also need to configure credentials to access modules.

6.11.4 Module Versions

Each module in the registry is versioned. These versions syntactically must follow semantic versioning.

6.12 Input Variables

Input variables let you customize aspects of Terraform modules without altering the module's own source code, making your module composable and reusable.

When you declare variables in child modules, the calling module should pass values in the **module** block.

6.12.1 Declaring an Input Variable

Each input variable accepted by a module must be declared using a **variable** block, eg:

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

The label after the **variable** keyword is a name for the variable, which must be unique among all variables in the same module.

6.12.2 Arguments

Terraform CLI defines the following optional arguments for variable declarations:

- **default** - A default value which then makes the variable optional.
- **type** - This argument specifies what value types are accepted for the variable.
- **description** - This specifies the input variable's documentation.
- **validation** - A block to define validation rules, usually in addition to type constraints.
- **sensitive** - Limits Terraform UI output when the variable is used in configuration.
- **nullable** - Specify if the variable can be null within the module.

6.12.3 Default values

the variable is considered to be optional and the default value will be used if no value is set when calling the module or running Terraform.
The **default** argument requires a literal value and cannot reference other objects in the configuration.

6.12.4 Type Constraints

The argument allows you to restrict the type of value that will be accepted as the value for a variable.
The keyword **any** may be used to indicate that any type is acceptable.

6.12.5 Input Variable Documentation

Because the input variables of a module are part of its user interface, you can briefly describe the purpose of each variable using this argument

6.12.6 Custom Validation Rules

A module author can specify arbitrary custom validation rules for a particular variable using a **validation** block

```
variable "image_id" {
  type        = string
  description = "The id of the machine image (AMI) to use for the server."

  validation {
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4)
    error_message = "The image_id value must be a valid AMI id, starting wi
  }
}
```

The condition argument is an expression that must use the value of the variable to return true or false
If **condition** evaluates to false, Terraform will produce an error message that includes the sentences given in **error_message**.
Multiple **validation** blocks can be declared in which case error messages will be returned for all failed conditions.

6.12.7 Suppressing Values in CLI Output

Setting a variable as **sensitive** prevents Terraform from showing its value in the **plan** or **apply** output, when you use that variable elsewhere in your configuration.
Terraform will still record sensitive values in the state, and so anyone who can access the state data will have access to the sensitive values in cleartext.
Any expressions whose result depends on the sensitive variable will be treated as sensitive themselves.
Any expressions whose result depends on the sensitive variable will be treated as sensitive themselves.
A provider can also declare an attribute as sensitive.
If you use a sensitive value from as part of an output value then Terraform will require you to also mark the output value itself as sensitive.

6.12.8 Disallowing Null Input Values

The **nullable** argument in a variable block controls whether the module caller may assign the value **null** to the variable.

6.12.9 Using Input Variable Values

Within the module that declared a variable, its value can be accessed from within expressions as **var.<NAME>**

Input variables are created by a **variable** block, but you reference them as attributes on an object named **var**.

To specify individual variables on the command line, use the **-var** option when running **plan** or **apply** commands.

You can use the **-var** option multiple times in a single command to set several different variables.

6.12.10 Variable Definitions Files

To set lots of variables, it is more convenient to specify their values in a variable definitions file (**.tfvars** or **.tfvars.json**) and then specify that file

on the command line with - **var-file**.
A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable.
Terraform also automatically loads variable definitions files if they are present

Variables can not use the meta-argument **depends_on**

Terraform variable name should always start with a letter

6.12.11 Environment Variables

Terraform searches the environment of its own process for environment variables named **TF_VAR_** followed by the name of a declared variable.

6.13 Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.
Resource instances managed by Terraform each export attributes whose values can be used elsewhere in configuration.
Output values are a way to expose some of that information to the user of your module.

6.13.1 Declaring an Output Value

Each output value exported by a module must be declared using an **output** block:

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

6.13.2 Accessing Child Module Outputs

In a parent module, outputs of child modules are available in expressions as: **module.<MODULE NAME>.<OUTPUT NAME>**

6.13.3 Optional Arguments

output blocks can optionally include **description**, **sensitive**, and **depends_on** arguments.

Because the output values of a module are part of its user interface, you can briefly describe the purpose of each value using the optional **description** argument.

An output can be marked as containing sensitive material using the optional **sensitive** argument.
Terraform will still record sensitive values in the state.

Since output values are just a means for passing data out of a module, it is usually not necessary to worry about their relationships with other nodes in the dependency graph.
depends_on argument can be used to create additional explicit dependencies.

6.14 Module Blocks

A module is a container for multiple resources that are used together.

Every Terraform configuration has at least one module, known as its root module.

6.14.1 Calling a Child Module

To call a module means to include the contents of that module into the configuration with specific values for its input variables.
Modules are called from within other modules using **module** blocks.

A module that includes a **module** block like this is the calling module of the child module.

The label immediately after the **module** keyword is a local name, which the calling module can use to refer to this instance of the module.

6.14.2 Source

All modules require a **source** argument, which is a meta-argument defined by Terraform.
Its value is either the path to a local directory containing the module's configuration files, or a remote module source that Terraform should download and use.

The same source address can be specified in multiple module blocks to create multiple copies of the resources defined within.

After adding, removing, or modifying **module** blocks, you must re-run terraform init to allow Terraform the opportunity to adjust the installed modules.

6.14.3 Version

When using modules installed from a module registry, we recommend explicitly constraining the acceptable version numbers to avoid unexpected or unwanted changes.

Use the **version** argument in the **module** block to specify versions

6.14.4 Meta-arguments

Along with source and version, Terraform defines a few more optional meta-arguments that have special meaning across all modules

Moving **resource** blocks from one module into several child modules causes Terraform to see the new location as an entirely different resource.

7 Read and write configuration

Terraform configuration uses the HashiCorp Configuration Language (HCL)

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects..

7.1 Resource Addressing

A resource address is a string that identifies zero or more resource instances in your overall configuration.
An address is made up of two parts:

```
[module path][resource spec]
```

7.1.1 Module path

A module path addresses a module within the tree of modules. It takes the form:

```
module.module\_name[module index]
```

An address without a resource spec, i.e. module.foo applies to every resource within the module if a single module, or all instances of a module if a module has multiple instances.

7.1.2 Resource spec

A resource spec addresses a specific resource instance in the selected module. It has the following syntax:

```
resource_type.resource_name[instance index]
```

7.2 References to Named Values

Terraform makes several kinds of named values available. Each of these names is an expression that references the associated value; you can use them as standalone expressions, or combine them with other expressions to compute new values.

7.2.1 Resources

<RESOURCE TYPE>.<NAME> represents a managed resource of the given type and name.
The value of a resource reference can vary, depending on whether the resource uses **count** or **for_each**.

Any named value that does not match another pattern listed below will be interpreted by Terraform as a reference to a managed resource.

7.2.2 Input Variables

var.<NAME> is the value of the input variable of the given name. If you define a variable as being of an object type with particular attributes then only those specific attributes will be available in expressions elsewhere in the module, even if the caller actually passed in a value with additional attributes.

7.2.3 Local Values

local.<NAME> is the value of the local value of the given name.
Local values can refer to other local values, even within the same **locals** block, as long as you don't introduce circular dependencies.

7.2.4 Child Module Outputs

module.<MODULE NAME> is an value representing the results of a module block.
To access one of the module's output values, use **module.<MODULE NAME>.<OUTPUT NAME>**

7.2.5 Data Sources

data.<DATA TYPE>.<NAME> is an object representing a data resource of the given data source type and name.

7.2.6 Filesystem and Workspace Info

7.2.7 Block-Local Values

Within the bodies of certain blocks, or in some other specific contexts, there are other named values available beyond the global values.

7.2.8 Named Values and Dependencies

Constructs like resources and module calls often use references to named values in their block bodies, and Terraform analyzes these expressions to automatically infer dependencies between objects.

7.2.9 References to Resource Attributes

The most common reference type is a reference to an attribute of a resource which has been declared either with a **resource** or **data** block

7.2.10 Sensitive Resource Attributes

When defining the schema for a resource type, a provider developer can mark certain attributes as sensitive, in which case Terraform will show a placeholder marker **(sensitive)** instead of the actual value when rendering a plan involving that attribute.

If you use a sensitive value from a resource attribute as part of an output value then Terraform will require you to also mark the output value itself as sensitive, to confirm that you intended to export it.

Terraform will still record sensitive values in the state, and so anyone who can access the state data will have access to the sensitive values in cleartext

7.2.11 Values Not Yet Known

When Terraform is planning a set of changes that will apply your configuration, some resource attribute values cannot be populated immediately because their values are decided dynamically by the remote system.
Each data source in turn belongs to a provider.

7.2.12 Local-only Data Sources

While many data sources correspond to an infrastructure object type that is accessed via a remote network API, some specialized data sources operate only within Terraform itself.
The behavior of local-only data sources is the same as all other data sources, but their result data exists only temporarily during a Terraform operation, and is re-calculated each time a new plan is created.

7.2.13 Data Resource Dependencies

Data resources have the same dependency resolution behavior as defined for managed resources.

7.3 Query data sources

To allow expressions to still be evaluated during the plan phase, Terraform uses special "unknown value" placeholders for these results.

Unknown values appear in the terraform plan output as **(not yet known)**.

7.4 Data Sources

Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

Each provider may offer data sources alongside its set of resource types.

A data source is accessed via a special kind of resource known as a data resource
The data source and name together serve as an identifier for a given resource and so must be unique within a module.

While managed resources cause Terraform to create, update, and delete infrastructure objects, data resources cause Terraform only to read objects.

7.4.1 Data Source Arguments

Each data resource is associated with a single data source, which determines the kind of object (or objects) it reads and what query constraint arguments are available.
Each data source in turn belongs to a provider,

7.4.2 Data Resource Behavior

If the query constraint arguments for a data resource refer only to constant values or values that are already known, the data resource will be read and its state updated during Terraform's "refresh" phase, which runs prior to creating a plan.
This ensures that the retrieved data is available for use during planning

Query constraint arguments may refer to values that cannot be determined until after configuration is applied, such as the id of a managed resource that has not been created yet.
In this case, reading from the data source is deferred until the apply phase

7.4.3 Local-only Data Sources

While many data sources correspond to an infrastructure object type that is accessed via a remote network API, some specialized data sources operate only within Terraform itself

The behavior of local-only data sources is the same as all other data sources, but their result data exists only temporarily during a Terraform operation, and is re-calculated each time a new plan is created.

7.4.4 Data Resource Dependencies

Data resources have the same dependency resolution behavior as defined for managed resources.

In order to ensure that data sources are accessing the most up to date information possible, arguments directly referencing managed resources are treated the same as if the resource was listed in **depends_on**.

This behavior can be avoided when desired by indirectly referencing the managed resource values through a **local** value.

7.4.5 Selecting a Non-default Provider Configuration

Data resources support the **provider** meta-argument as defined for managed resources

7.4.6 Description

The **data** block creates a data instance of the given type and name. The combination of the type and name must be unique.
Within the block (the **{ }**) is configuration for the data instance. The configuration is dependent on the type.
Each data instance will export one or more attributes, which can be used in other resources as reference expressions of the form:
data. **<TYPE>.<NAME>.<ATTRIBUTE>**

7.5 Resource Graph

Terraform builds a dependency graph from the Terraform configurations, and walks this graph to generate plans, refresh state, and more.

7.5.1 Graph Nodes

There are only a handful of node types that can exist within the graph:

- **Resource Node** Represents a single resource
- **Provider Configuration Node** Represents the time to fully configure a provider. This is when the provider configuration block is given to a provider
- **Resource Meta-Node** Represents a group of resources, but does not represent any action on its own. This is done for convenience on dependencies and making a prettier graph

When visualizing a configuration with **terraform graph**, you can see all of these nodes present

7.5.2 Building the Graph

Building the graph is done in a series of sequential steps:

- Resources nodes are added based on the configuration. If a diff (plan) or state is present, that meta-data is attached to each resource node
- Resources are mapped to provisioners if they have any defined. This must be done after all resource nodes are created so resources with the same provisioner type can share the provisioner implementation
- Explicit dependencies from the **depends_on** meta-parameter are used to create edges between resources.
- If a state is present, any "orphan" resources are added to the graph. Orphan resources are any resources that are no longer present in the configuration but are present in the state file.
- Resources are mapped to providers. Provider configuration nodes are created for these providers, and edges are created such that the resources depend on their respective provider
- Interpolations are parsed in resource and provider configurations to determine dependencies. References to resource attributes are turned into dependencies from the resource with the interpolation to the resource being referenced
- Create a root node. The root node points to all resources and is created so there is a single root to the dependency graph
- If a diff is present, traverse all resource nodes and find resources that are being destroyed. These resource nodes are split into two: one node that destroys the resource and another that creates the resource
- Validate the graph has no cycles and has a single root.

7.5.3 Walking the Graph

To walk the graph, a standard depth-first traversal is done. Graph walking is done in parallel: a node is walked as soon as all of its dependencies are walked.

The amount of parallelism is limited using a semaphore to prevent too many concurrent operations from overwhelming the resources of the machine running Terraform.

7.6 Complex Types

Terraform module authors and provider developers can use detailed type constraints to validate user-provided values for their input variables and resource arguments.
This requires some additional knowledge about Terraform's type system, but allows you to build a more resilient user interface for your modules and resources

7.6.1 Primitive Types

A primitive type is a simple type that isn't made from any other types. All primitive types in Terraform are represented by a type keyword.

7.6.2 Conversion of Primitive Types

The Terraform language will automatically convert **number** and **bool** values to **string** values when needed, and vice-versa as long as the string contains a valid representation of a number or boolean value.

7.6.3 Complex Types

A complex type is a type that groups multiple values into a single value. There are two categories of complex types: collection types and structural types.

A collection type allows multiple values of one other type to be grouped together as a single value. The type of value within a collection is called its element type. All collection types must have an element type.

A structural type allows multiple values of several distinct types to be grouped together as a single value. Structural types require a schema as an argument, to specify which types are allowed for which elements.

7.6.4 Conversion of Complex Types

Similar kinds of complex types can usually be used interchangeably, and most of Terraform's documentation glosses over the differences between the kinds of complex type. This is due to two conversion behaviors:

- Whenever possible, Terraform converts values between similar kinds of complex types if the provided value is not the exact type requested.
- Whenever possible, Terraform converts element values within a complex type, by converting complex-typed elements recursively

7.6.5 Dynamic Types: The "any" Constraint

The keyword **any** is a special construct that serves as a placeholder for a type yet to be decided. **any** is not itself a type: when interpreting a value against a type constraint containing **any**, Terraform will attempt to find a single actual type that could replace the **any** keyword to produce a valid result.

All of the elements of a collection must have the same type, so conversion to **list(any)** requires that all of the given elements must be convertible to a common type.

If you wish to apply absolutely no constraint to the given value, the **any** keyword can be used in isolation.

7.6.6 Experimental: Optional Object Type Attributes

there is experimental support for marking particular attributes as optional in an object type constraint.
To mark an attribute as optional, use the additional **optional(...)** modifier around its type declaration.

Terraform will return an error if the source value has no matching attribute. Marking an attribute as optional changes the behavior in that situation Terraform will instead just silently insert **null** as the value of the attribute, allowing the receiving module to describe an appropriate fallback behavior.

7.7 Built-in Functions

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

```
max(5, 12, 9)
```

The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use.

8 Manage state

Terraform uses state to keep track of the infrastructure it manages. To use Terraform effectively, you have to keep your state accurate and secure.

8.1 State Locking

If supported by your backend, Terraform will lock your state for all operations that could write state. This prevents others from acquiring the lock and potentially corrupting your state.
State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, Terraform will not continue. You can disable state locking for most commands with the **-lock** flag.

The Terraform lock is writted under the **.terraform.lock.hcl** file.

If acquiring the lock is taking longer than expected, Terraform will output a status message. If Terraform doesn't output a message, state locking is still occurring.

Not all backends support locking.
Terraform perform state locking by default

8.1.1 Force Unlock

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

To protect you, the **force-unlock** command requires a unique lock ID. Terraform will output this lock ID if unlocking fails. This lock ID acts as a nonce, ensuring that locks and unlocks target the correct lock.

9 Sensitive Data in State

Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive".

When using local state, state is stored in plain-text JSON files.

When using remote state, state is only ever held in memory when used by Terraform. It may be encrypted at rest, but this depends on the specific remote state backend.

If you manage any sensitive data with Terraform treat the state itself as sensitive data.

Storing state remotely can provide better security. Terraform does not persist state to the local disk when remote state is in use, and some backends can be configured to encrypt the state data at rest.

9.1 Refresh-Only Mode

Refresh-only mode instructs Terraform to create a plan that updates the Terraform state to match changes made to remote objects outside of Terraform. This is useful if state drift has occurred and you want to reconcile your state file to match the drifted remote objects.

9.2 Command: login

The **terraform login** command can be used to automatically obtain and save an API token for Terraform Cloud, Terraform Enterprise, or any other

host that offers Terraform services.

If you don't provide an explicit hostname, Terraform will assume you want to log in to Terraform Cloud at **app.terraform.io.**

9.2.1 Credentials Storage

Terraform will obtain an API token and save it in plain text in a local CLI configuration file called **credentials.tfrc.json.** When you run **terraform login**, it will explain specifically where it intends to save the API token and give you a chance to cancel.

If you don't wish to store your API token in the default location, you can optionally configure a credentials helper program which knows how to store and later retrieve credentials in some other system.

The **terraform login** command works with any server supporting the login protocol

9.3 Backends

Backends define where Terraform's state snapshots are stored.
A given Terraform configuration can either specify a backend, integrate with Terraform Cloud, or do neither and default to storing state locally.

9.3.1 What Backends Do

Backends primarily determine where Terraform stores its state. Terraform uses this persisted state data to keep track of the resources it manages.
By default, Terraform implicitly uses a backend called local to store state as a local file on disk. Every other backend stores state in a remote service of some kind, which allows multiple people to access it.

The built-in backends are the only backends. You cannot load additional backends as plugins.

9.4 Local Backends

The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally. The two argument available for local backends are: **path** and **workspace-dir**

9.4.1 Backend Configuration

Each Terraform configuration can specify a backend, which defines where state snapshots are stored.
If your configuration includes a **cloud** block, it cannot include a **backen** block.

9.4.2 Using a Backend Block

Backends are configured with a nested **backend** block within the top-level **terraform** block: There are some important limitations on backend configuration:

- configuration can only provide one backend block
- backend block cannot refer to named values

9.4.3 Backend Types

The block label of the backend block ("remote", in the example above) indicates which backend type to use, and the configured backend must be available in the version of Terraform you are using.

The arguments used in the block's body are specific to the chosen backend type; they configure where and how the backend will store the configuration's state, and in some cases configure other behavior

In normal use we do not recommend including access credentials as part of the backend configuration. Instead, leave those arguments completely unset and provide credentials via the credentials files or environment variables

9.4.4 Initialization

Whenever a configuration's backend changes, you must run **terraform init** again to validate and configure the backend before you can perform any plans,

applies, or state operations.
When changing backends, Terraform will give you the option to migrate your state to the new backend.

9.4.5 Partial Configuration

When some or all of the arguments are omitted, we call this a partial configuration.
With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process.
If backend settings are provided in multiple locations, the top-level settings are merged such that any command-line options override the settings in the main configuration and then the command-line options are processed in order.

The final, merged configuration is stored on disk in the **.terraform** directory, which should be ignored from version control.

9.4.6 File

A backend configuration file has the contents of the **backend** block as top-level attributes, without the need to wrap it in another **terraform** or **backend** block

9.4.7 Changing Configuration

You can change your backend configuration at any time. You can change both the configuration itself as well as the type of backend.
Terraform will automatically detect any changes in your configuration and request a reinitialization. As part of the reinitialization process, Terraform will ask if you'd like to migrate your existing state to the new configuration.

If you're just reconfiguring the same backend, Terraform will still ask if you want to migrate your state.

9.4.8 Unconfiguring a Backend

If you no longer want to use any backend, you can simply remove the configuration from the file. Terraform will detect this like any other change and prompt you to reinitialize

9.4.9 Terraform Cloud Configuration

The main module of a Terraform configuration can integrate with Terraform Cloud to enable its CLI-driven run workflow. You only need to configure these settings when you want to use Terraform CLI to interact with Terraform Cloud.

To configure the Terraform Cloud CLI integration, add a nested **cloud** block within the **terraform** block. You cannot use the CLI integration and a state backend in the same configuration.

9.5 Create a workspace

Once you have created a Terraform Cloud account and created or joined an organization, you can start managing infrastructure with Terraform Cloud.

10 Understand Terraform Cloud

Terraform Cloud helps teams collaborate on Infrastructure as Code by providing a stable and reliable environment for operations, shared state and secret data, access controls to manage permissions for team members, and a policy framework for governance.

10.1 What is Terraform Cloud?

Terraform Cloud is an application that helps teams use Terraform together. It manages Terraform runs in a consistent and reliable environment, and includes easy access to shared state and secret data, access controls for approving changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations, and more.

10.2 Terraform Workflow

Terraform Cloud runs Terraform CLI to provision infrastructure.

In its default state, Terraform CLI uses a local workflow, performing operations on the workstation where it is invoked and storing state in a local directory.

Since teams must share responsibilities and awareness to avoid single points of failure, working with Terraform in a team requires a remote workflow. At minimum, state must be shared; ideally, Terraform should execute in a consistent remote environment.
Terraform Cloud offers a team-oriented remote Terraform workflow, designed to be comfortable for existing Terraform users and easily learned by new users.

10.2.1 Remote Terraform Execution

Terraform Cloud runs Terraform on disposable virtual machines in its own cloud infrastructure by default.
Remote Terraform execution is sometimes referred to as "remote operations."

Remote execution helps provide consistency and visibility for critical provisioning operations.

10.2.2 Organize Infrastructure with Projects and Workspaces

Terraform's local workflow manages a collection of infrastructure with a persistent working directory, which contains configuration, state data, and variables.
Terraform Cloud organizes infrastructure into projects and workspaces instead of directories. Each workspace contains everything necessary to manage a given collection of infrastructure.

You can use projects to organize your workspaces into groups.
This lets you grant access to collections of workspaces instead of using workspace-specific or organization-wide permissions, making it easier to limit access to only the resources required for a team member's job function.

10.2.3 Remote State Management, Data Sharing, and Run Triggers

Terraform Cloud acts as a remote backend for your Terraform state. State storage is tied to workspaces, which helps keep state associated with the configuration that created it.
Only remote backend can perform remote run on on-premise infrastructure or in Terraform Clouds.

Terraform Cloud also enables you to share information between workspaces with root-level outputs. Separate groups of infrastructure resources often need to share a small amount of information, and workspace outputs are an ideal interface for these dependencies.

Workspaces that use remote operations can use **terraform_remote_state** data sources to access other workspaces' outputs, subject to per-workspace access controls.
And since new information from one workspace might change the desired infrastructure state in another, you can create workspace-to-workspace run triggers to ensure downstream workspaces react when their dependencies change.
Backend types doesn't support state locking: **artifactory**, **etcd**

10.2.4 Version Control Integration

Like other kinds of code, infrastructure-as-code belongs in version control.
Each workspace can be linked to a VCS repository that contains its Terraform configuration, optionally specifying a branch and subdirectory. Terraform Cloud automatically retrieves configuration content from the repository, and will also watch the repository for changes.

VCS integration is powerful, but optional; if you use an unsupported VCS or want to preserve an existing validation and deployment pipeline, you can use the API or Terraform CLI to upload new configuration versions.

10.2.5 Command Line Integration

Remote execution offers major benefits to a team, but local execution offers major benefits to individual developers.

Terraform Cloud offers the best of both worlds, allowing you to run remote plans from your local command line.

Remote CLI-driven runs use the current working directory's Terraform configuration and the remote workspace's variables, so you don't need to obtain production cloud credentials just to preview a configuration change.

10.2.6 Private Registry

Terraform can fetch providers and modules from many sources. Terraform Cloud makes it easier to find providers and modules to use with a private registry. Users throughout your organization can browse a directory of internal providers and modules, and can specify flexible version constraints for the modules they use in their configurations.

The private registry uses your VCS as the source of truth, relying on Git tags to manage module versions

10.2.7 Integrations

In addition to providing powerful extensions to the core Terraform workflow, Terraform Cloud makes it simple to integrate infrastructure provisioning with your business's other systems.

10.2.8 Full API

Nearly all of Terraform Cloud's features are available in its API, which means other services can create or configure workspaces, upload configurations, start Terraform runs, and more.

10.2.9 Notifications

Terraform Cloud can send notifications about Terraform runs to other systems, including Slack and any other service that accepts webhooks.

10.2.10 Run Tasks

Run Tasks allow Terraform Cloud to execute tasks in external systems at specific points in the Terraform Cloud run lifecycle.

10.2.11 Access Control and Governance

Larger organizations are more complex, and tend to use access controls and explicit policies to help manage that complexity. Terraform Cloud's paid upgrade plans provide extra features to help meet the control and governance needs of large organizations.

10.2.12 Team-Based Permissions System

With Terraform Cloud's team management, you can define groups of users that match your organization's real-world teams and assign them only the permissions they need

10.2.13 Policy Enforcement

Policy-as-code lets you define and enforce granular policies for how your organization provisions infrastructure. You can limit the size of compute VMs, confine major updates to defined maintenance windows, and much more.

10.2.14 Cost Estimation

Before making changes to infrastructure in the major cloud providers, Terraform Cloud can display an estimate of its total cost, as well as any change in cost caused by the proposed updates.

10.3 Workspaces

Working with Terraform involves managing collections of infrastructure resources, and most organizations manage many different collections.

Terraform Cloud manages infrastructure collections with workspaces instead of directories. A workspace contains everything Terraform needs to manage a given collection of infrastructure, and separate workspaces function like completely separate working directories.

In addition to the basic Terraform content, Terraform Cloud keeps some additional data for each workspace:

- **State versions**: Each workspace retains backups of its previous state files.

- **Run history**: When Terraform Cloud manages a workspace's Terraform runs, it retains a record of all run activity, including summaries, logs,

The top of each workspace shows a resource count, which reflects the number of resources recorded in the workspace's state file.

Workspace local state file are saved under the **terraform.tfstated.d** directory

To create a new workspace use the following command:
terraform workspace new <workspace_name>

To switch between different workspaces use the following command: **terraform workspace select <workspace_name>**

10.3.1 Terraform Runs

For workspaces with remote operations enabled, Terraform Cloud performs Terraform runs on its own disposable virtual machines, using that workspace's configuration, variables, and state.

10.3.2 Workspace Health

Terraform Cloud can perform automatic health assessments in a workspace to assess whether its real infrastructure matches the requirements defined in its Terraform configuration.

- **Drift detection** determines whether your real-world infrastructure matches your Terraform configuration.

- **Continuous validation** determines whether custom conditions in the workspace's configuration continue to pass after Terraform provisions the infrastructure.

You can enforce health assessments for all eligible workspaces or let each workspace opt in to health assessments through workspace settings.

10.4 Use Modules from the Registry

Modules have output values. You can reference them with the:
module.MODULE_NAME.OUTPUT_NAME
naming convention.

When using a new module for the first time, you must run either **terraform init** or **terraform get** to install the module

10.5 Private Registry

Terraform Cloud's private registry works similarly to the public Terraform Registry and helps you share Terraform providers and Terraform modules across your organization.

10.5.1 Public Providers and Modules

Public modules and providers are hosted on the public Terraform Registry and Terraform Cloud can automatically synchronize them to an organization's private registry

10.5.2 Private Providers and Modules

Private providers and private modules are hosted on an organization's private registry and are only available to members of that organization. In Terraform Enterprise, private providers and modules are also available to other organizations that are configured to share with that organization.

10.5.3 Managing Usage

You can create Sentinel policies to manage how members of your organization can use modules from the private registry.

10.6 Terraform Cloud Teams

Teams are groups of Terraform Cloud users within an organization. If a user belongs to at least one team in an organization, they are considered a member of that organization.

The organization can grant workspace permissions to teams that allow its members to start Terraform runs, create workspace variables, read and write state, etc...
Teams can only have permissions on workspaces within their organization,

10.6.1 The Owners Team

Every organization has a team named owners. An organization's creator is the first member of its owners team, you can add and remove other members in the same way as other teams.

Unlike other teams, you cannot delete the owners team or leave it empty. If there is only one member in an owners team, you must add another user before you can remove the current member.

10.6.2 Managing Teams

Only organization owners can create teams, assign teams permissions, or view the full list of teams. Other users can view any teams marked as visible within the organization, plus any secret teams they are members of.

10.6.3 Managing Workspace Access

You can grant teams various permissions on workspaces.
Terraform Cloud uses the highest permission level from your teams to determine what actions you can take on a particular resource.

To switch between different workspaces you can use the **terraform workspace select <name>**

10.7 Defining Sentinel Policies

Policies are rules that Terraform Cloud enforces on runs. You use the Sentinel policy language to define Sentinel policies. After you define policies, you must add them to policy sets that Terraform Cloud can enforce globally or on specific projects and workspaces.

10.7.1 Sentinel Imports

A policy can include imports that enable a policy to access reusable libraries, external data, and functions.
Terraform Cloud provides four imports to define policy rules for the plan, configuration, state, and run associated with a policy check.

10.7.2 Useful Functions and Idioms for Terraform Sentinel Policies

The following functions and idioms will be useful as you start writing Sentinel policies for Terraform.

Iterate over Modules and Find Resources

The most basic Sentinel task for Terraform is to enforce a rule on all resources of a given type. Before you can do that, you need to get a collection of all the relevant resources from all modules.

Validate Resource Attributes
Once you have a collection of resources instances of a desired type indexed by their addresses, you usually want to validate that one or more resource attributes meets some conditions by iterating over the resource instances.

While you could use Sentinel's all and any expressions directly inside Sentinel rules, your rules would only report the first violation because Sentinel uses short-circuit logic. It is therefore usually preferred to use a for loop outside of your rules so that you can report all violations that occur. You can do this inside functions or directly in the policy itself.

Sentinel function prints a warning message for every resource instance that violates the condition. This allows writers of Terraform code to fix all violations after just one policy check.

While functions allows a rule to validate an attribute against a list, some rules will only need to validate an attribute against a single value; in those cases, you could either use a list with a single value or embed that value inside the function itself.

10.7.3 Validate Multiple Conditions in a Single Policy

If you want a policy to validate multiple conditions against resources of a specific type, you could define a separate validation function for each condition or use a single function to evaluate all the conditions. In the latter case, you would make this function return a list of boolean values, using one for each condition.

10.8 Enforce a policy

Sentinel is an embedded policy-as-code framework integrated with various HashiCorp products. It enables fine-grained, logic-based policy decisions, and can use information from external sources. Terraform Cloud lets users enforce Sentinel policies as part of the run workflow.

A policy consists of:

- The policy controls defined as code.
- An enforcement level that determines run behavior in the event of policy failure.

To apply a policy to a workspace and it's run, you must first add it to a policy set. Each policy set can apply to specific workspaces, or to all workspaces within an organization.

To create a policy set, you will need a VCS repository to host the policy configuration.

10.8.1 Explore a policy set

Enforcement levels establish whether or not an operation can proceed if a policy fails. Sentinel provides three enforcement levels:

- **Hard-mandatory** requires that the policy passes. If a policy fails, the run stops. You must resolve the failure to proceed.
- **Soft-mandatory** lets an organization owner or a user with override privileges proceed with the run in the event of failure. Terraform Cloud logs all overrides.

- **Advisory** will notify you of policy failures, but proceed with the operation.

Sentinel code files must follow the naming convention of **<policy name>.sentinel**

Policy set names within a Terraform Cloud organization must be unique