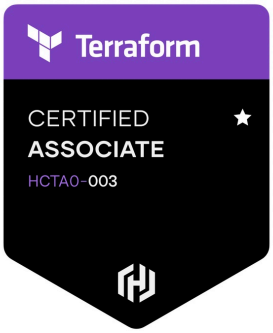
Terraform Associate 003 Certification Study Guide



Content:

Note: This study guide is based on the official terraform documentation for Terraform Associate 003 Certification.

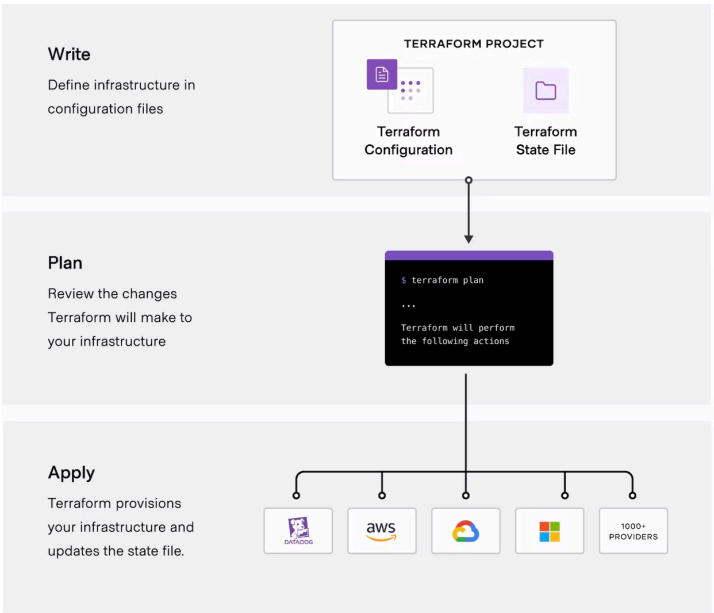
Author: Esaú Reyes Valdespino

(1) Understand Infrastructure as Code (IaC) concepts

- 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.

- The core Terraform workflow consists of three stages:

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



- Terraform **tracks your infrastructure in a state file**, which acts as a **source of truth** for your environment.
- Terraform **configuration files are declarative,** meaning that they describe the end state of your infrastructure.
- Supports reusable configuration components called modules that define configurable collections of infrastructure, you can use publicly available modules from Terraform Registry, or write your own.
- Terraform Cloud runs Terraform in a consistent, reliable environment and provides secure access to shared state and secret data, role-based access controls, a private registry for sharing both modules and providers, and more.
- The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remain the same.

(2) Understand the purpose of Terraform (vs other IaC)

- Terraform let you use the same workflow to manage multiple providers and handle cross-cloud dependencies.
- Efficiently deploy, release, scale and monitor infrastructure multi-tier applications. Terraform allows you to manage the resources in each tier together, and automatically handles dependencies between tiers.
- Terraform Cloud can also integrate with ticketing systems like **ServiceNow** to **automatically generate infrastructure requests**.
- Terraform can help you enforce policies on the types of resources teams can provision and use. **Sentinel, a policy-as-code framework,** to automatically enforce compliance and governance policies before Terraform makes infrastructure changes (Available on Terraform Enterprise and Terraform Cloud).
- Network Infrastructure Automation (NIA) allows you to safely approve the changes that your applications require without having to manually translate tickets from developers into the changes you think their applications need.
- Terraform lets you rapidly spin up and decommission infrastructure for development, test, QA, and production. Using Terraform to create disposable environments as needed is more cost-effective than maintaining each one indefinitely.
- Terraform can guarantee a one-to-one mapping when it creates objects and records their identities in the state.
- When importing objects created outside of Terraform, you must make sure that each distinct object is imported to only one resource instance.
- To ensure correct operation, Terraform retains a copy of the most recent set of dependencies within the state.
- In addition to basic mapping, T**erraform 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.
- This is de default behavior of Terraform: for every **plan and apply Terraform will sync all resources in your state.**
- For larger infrastructures, querying every resource is too slow. Larger users of Terraform make heavy use of the **-refresh=false** flag as well as the **-target** flag in order to work around this. In this scenarios, the cached state is treated as the record of truth.
- As default Terraform stores the state file in the current working directory where Terraform was run.
- Remote state is the recommended solution. With a fully-featured state backend, Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

(3) Understand Terraform basics

- Providers allow Terraform to interact with cloud providers, SaaS providers, and other APIs.
- **Provider configuration** belong in the **root module** of a Terraform configuration.
- A provider's documentation should list which configuration arguments it expects.
- There are two meta-arguments that are defined by Terraform itself and available for all providers blocks:
  - **alias:** for using the same provider with different configurations for different resources.
  - **version:** which we no longer recommend (use provider requirements instead).
- A provider block may be omitted if its contents would be empty. Terraform assumes an empty default configuration for any provider.
- You can optionally define **multiple configurations for the same provider**, and select which one to use on a **per-resource or per-module** basis. The primary reason for this is to support **multiple regions** for a cloud platform.
- To create multiple configurations for a given provider, include multiple provider blocks with the same provider name and use **"alias"** for each **non-default configuration**.
- Resources that don't set the provider meta-argument will use the default provider configuration that matches the first word of the resource type name.pwd
- The special **"terraform"** configuration block type is used to configure some behaviors of terraform itself, such as requiring a minimum Terraform version to apply you configuration. Within a **"terraform"** block only constant values can be used.
- The **nested "cloud" block** configures Terraform Cloud for enabling its **CLI-driven run workflow.**
- The **nested "backend"** block configures which state backend Terraform should use.
- The **"required_version"** setting accepts a version constraint string, which specifies which versions of Terraform can be used with your configuration. Child modules can specify its own version requirements. This setting applies only to the version of Terraform CLI.
- The **"required_providers"** block specifies all of the providers required by the current module, mapping each local provider name to a source address and a version constraint.
- You can enable experimental features when they are available in your current releases by setting the **"experiments"** argument inside a terraform block. Any module with experiments enabled will generate a warning features in a every "terraform plan" or "terraform apply".
- The **"terraform"** block have a nested **"provider_meta"** block for each provider module is using, if the provider defines a schema for it.
- A **Terraform configuration** may refer to tow kinds of external dependency that come from outside of its own codebase:
  - **Providers: which are plugins for Terraform** that extend it with support for interacting with various external systems.
  - **Modules:** which allow splitting out groups of Terraform configuration constructs into usable abstractions.

- Version constraints within the configuration itself determine which version of dependencies are potentially compatible, but after selecting a specific version of each dependency Terraform remembers the decisions it made in a dependency lock file. Tracks only provider dependencies, Terraform does not remember version selections for remote modules, and so Terraform will always select the newest available module version that meets the specified version constraints. You can specify a exact version.
- The dependency lock file that belongs to the configuration as a whole, rather than to each separate module in the configuration. The **lock file** is always named **.terraform.lock.hcl**
- If a particular provider already has a selection recorded in the lock file, Terraform will always re-select that version for installation, even if a newer version has become available. You can override that behavior by adding **-upgrade** option when you run **"terraform init"**.
- When you add a new provider for the first time you can verify it in whatever way you choose, and then trust that Terraform will raise an error if a future run of terraform init encounters a non-matching package for the same provider version.
- **Terraform is logically split** into two main parts:
  - **Terraform Core:** is a statically-compiled binary written in the **Go programming language**. The compiled binary is the **command line tool CLI terraform**, the entry point for anyone using Terraform.
    - Infrastructure as Code
    - Resource state management
    - Construction of the Resource Graph
    - Plan execution
    - Communication with plugins over RPC
  - **Terraform Plugins:** are **written in Go** and are executables binaries invoked by Terraform Core over RPC. Each plugin exposes an implementation for a specific service, such as AWS, or provisioner, such as bash.
    - Initialization of any included libraries used to make API calls.
    - Authentication with the Infrastructure Provider.
    - Define Resources that map to specific Services.

(4) Use Terraform outside the core workflow

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

  terraform import [options] ADDRESS ID

  - Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS. The import command can import resources **into modules as well as directly into the root** of your state.
  - ID is dependent on the resource type being imported
  - It can only import **one resource at a time.**
  - An import may also result in a **"complex import"** where **multiple resources are imported**. It is necessary to consult the import output and create a resource block in configuration for each secondary resource.

The command-line flags are all optional. The list of available flags are:

- `-config=path` - Path to directory of Terraform configuration files that configure the provider for import. This defaults to your working directory. If this directory contains no Terraform configuration files, the provider must be configured via manual input or environmental variables.
- `-input=true` - Whether to ask for input for provider configuration.
- `-lock=false` - Don't hold a state lock during the operation. This is dangerous if others might concurrently run commands against the same workspace.
- `-lock-timeout=0s` - Duration to retry a state lock.
- `-no-color` - If specified, output won't contain any color.
- `-parallelism=n` - Limit the number of concurrent operation as Terraform walks the graph. Defaults to 10.
- `-provider=provider` - **Deprecated** Override the provider configuration to use when importing the object. By default, Terraform uses the provider specified in the configuration for the target resource, and that is the best behavior in most cases.
- `-var 'foo=bar'` - Set a variable in the Terraform configuration. This flag can be set multiple times. Variable values are interpreted as literal expressions in the Terraform language, so list and map values can be specified via this flag. This is only useful with the `-config` flag.
- `-var-file=foo` - Set variables in the Terraform configuration from a variable file. If a `terraform.tfvars` or any `.auto.tfvars` files are present in the current directory, they will be automatically loaded. `terraform.tfvars` is loaded first and the `.auto.tfvars` files after in alphabetical order. Any files specified by `-var-file` override any values set automatically from files in the working directory. This flag can be used multiple times. This is only useful with the `-config` flag.

- 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 instead. This command is a nested subcommand, meaning that it has further subcommands.

  terraform state <subcommand> [options] [args]

  - The Terraform state subcommands all work with remote state just as if it was local state.
  - All Terraform state subcommands that modify the state write backup files. The path of these backup files can be controlled with **-backup**

- Terraform has a detailed logs which can be enabled by setting the **TF_LOG** environment variable to nay value. This will cause detailed logs to appear on stderr.

  - You can set **TF_LOG** to one of the log levels **TRACE, DEBUG, INFO, WARN** or **ERROR**.
  - Logging can be enabled separately for terraform itself and the provider plugins using the **TF_LOG_CORE** or **TF_LOG_PROVIDER** environment variables. These takes the same level arguments as TF_LOG, but only activate a subset of the logs.
  - **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.
  - **TF_LOG** must be set in order for any logging to be enabled.

(5) Interact with Terraform modules

- The terraform registry is directly into Terraform, so a Terraform configuration can refer to any module published in the registry.

  <NAMESPACE>/<NAME>/<PROVIDER>

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

- You can also use modules from a private registry.

  <HOSTNAME>/<NAMESPACE>/<NAME>/<PROVIDER>

Depending on the registry you're using, you might also need to configure credentials to access modules. Each module in the registry is versioned.

- **Input variables** let you customize aspects of Terraform modules without altering the module's own source code. This functionality allows you to share modules across different Terraform configurations, making your modules composable and reusable.
  - When you declare variables in the root module of your configuration, you can set their values using CLI options and environments variables. When you declare them in the child modules, the calling module should pass values in the "module" block.
  - The name of the variable can be any valid identifier except the following: source, version, provider, count, for_each, lifecycle, depends_on, locals.

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

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

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

- Terraform CLI declares 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.

- Type constraints are created from a mixture of type keywords and type constructors. The supported type keywords are:

  - `string`
  - `number`
  - `bool`

  The type constructors allow you to specify complex types such as collections:

  - `list(<TYPE>)`
  - `set(<TYPE>)`
  - `map(<TYPE>)`
  - `object({<ATTR NAME> = <TYPE>, ... })`
  - `tuple([<TYPE>, ...])`

- The keyword **"any"** may be used to indicate that **any type is acceptable**.
- If both the **"type" and "default"** arguments are specified, the given default must be **convertible to the specified type**.
- Setting a variable as **"sensitive"** prevents Terraform from sharing its value in the plan or apply output, Terraform 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 expression whose result depends on the sensitive variable will be treated as sensitive themselves.
- Setting "nullable" to "false" ensures that the variable value will never be "null" within the module. If "nullable" is "false" and the variable has a default value, Terraform uses the default when a module input argument is null.
- Input variables are created by a "variable" block, but you reference them as attributes on an object named "var".
- When **variables** are declared in the **root module** of your configuration, they can be set in a number of ways:
  - In a Terraform **Cloud workspace**
  - Individually, with the **-var command** line option
  - In variable definitions **(.tfvars) files**, either specified on the command line or automatically loaded.
  - As **environment variables**.

- Terraform automatically loads a number of variables definitions files if they are present:
  - Files named exactly terraform.tfvars or terraform.tfvars.json
  - Any files with names ending in .auto.tfvars or .auto.tfvars.json
- Terraform searches the environment of its own process for **environment variables** named **"TF_VAR_"** followed by the name of a declared variable.
- Variable definition precedence: If the same variable is assigned multiple values, Terraform uses the last value it finds, overriding any previous values. Terraform **loads variables in the following order**, with later resource taking precedence over earlier ones:
  - Environment variables
  - The terraform.tfvars file, if present
  - The terraform.tfvars.json file, if present
  - Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
  - Any -var and -var-file options on the command line, in the order they are provided.

- **Module Output Values.**
  - The resources defined in a module are encapsulated, so the calling module cannot access their attributes directly. However, the child module can declare output values to selectively export certain values to be accessed by the calling module.
- **Transferring Resource State Into Modules**
  - Moving resource blocks from one module into several child modules causes Terraform to see the new location as an entirely different resource. As a result, Terraform plans to destroy all resources instances at the old address and create new instances at the new address.
  - To preserve existing objects, you can use **refactoring blocks to record the old new address for each resource instance.**
- **Replacing resources within a module**
  - You may have an object that needs to be replaced with a new object for a reason that isn't automatically visible to Terraform. In that case you can use the "**-replace=...**" planning option to force Terraform to propose replacing that object.

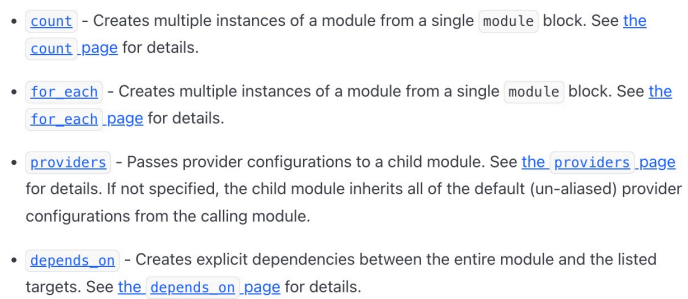
terraform plan -replace=module.example.aws_instance.example

- **Calling a Chill Module**
  - To call a module means to include the content of that module into the configuration with specific values for it input variables. **Modules are called** from within other modules **using "module" blocks.**

```
module "servers" {
  source = "./app-cluster"

  servers = 5
}
```

    - **Source:** All module 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 source that Terraform should download and use.
    - After **adding, removing, or modifying module blocks, you must re-run "terraform init"** to allow Terraform the opportunity to adjust the installed modules. By default this command will not upgrade an already-installed module; use the **-upgrade** option to instead upgrade to the newest available version.
    - When using modules installed from a module registry, we recommend explicitly constraining the acceptable version number to avoid unexpected or unwanted changes.

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

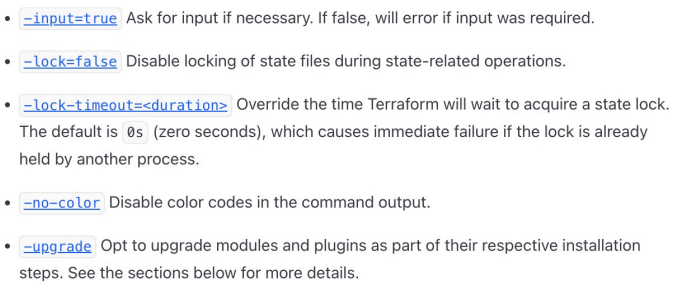


- `count` - Creates multiple instances of a module from a single `module` block. See the `count` page for details.
- `for_each` - Creates multiple instances of a module from a single `module` block. See the `for_each` page for details.
- `providers` - Passes provider configurations to a child module. See the `providers` page for details. If not specified, the child module inherits all of the default (un-aliased) provider configurations from the calling module.
- `depends_on` - Creates explicit dependencies between the entire module and the listed targets. See the `depends_on` page for details.

(6) Use the core Terraform workflow

- **The Core Terraform Workflow**
  - Write: Author infrastructure as code.
  - Plan: Preview changes before applying.
  - Apply: Provision reproducible infrastructure.
- **Write - Terraform Cloud**
  - Provides a centralized and secure location for storing input variables and state while also bringing back a tight feedback loop for speculative plans for config authors. Terraform configuration can interact with Terraform Cloud through the CLI integration.
  - After configure the integration, a Terraform Cloud API key is all your team members need to edit config and run speculative plans against the latest version of the state file using all the remotely stored input variables.
- **Plan - Terraform Cloud**
  - Once a pull request is ready for review, Terraform Cloud makes the process of reviewing a speculative plan easier for team members. First the plan is automatically run when the pull request es created.
  - Once the plan is complete, the status update indicates wether there were any changes in the speculative plan, right from the pull request view.
- **Apply - Terraform Cloud**
  - After merge, Terraform Cloud present the concrete plan to the team for review and approval.
  - The team can discuss any outstanding questions about the plan before the change is made.
  - Once the Apply is confirmed, Terraform Cloud displays the process live to anyone who'd like to watch.

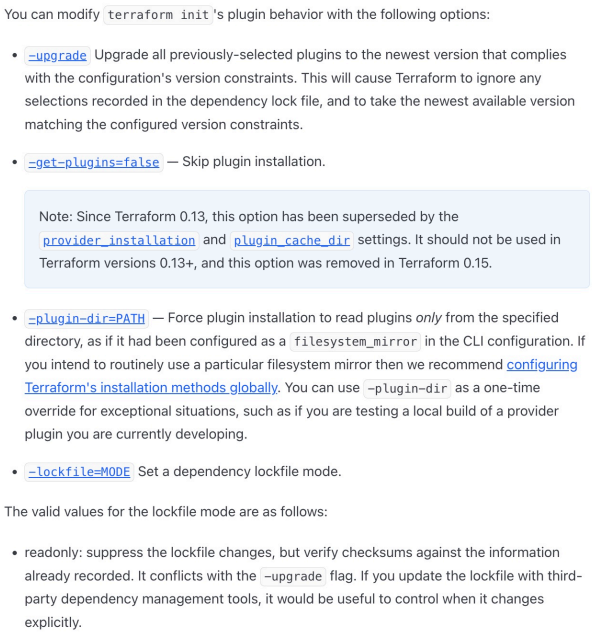
- **Command: init**
  - Is used to initialize a working directory containing Terraform configuration files. It's safe to run this command multiple times.



- `-input=true` Ask for input if necessary. If false, will error if input was required.
- `-lock=false` Disable locking of state files during state-related operations.
- `-lock-timeout=<duration>` Override the time Terraform will wait to acquire a state lock. The default is `0s` (zero seconds), which causes immediate failure if the lock is already held by another process.
- `-no-color` Disable color codes in the command output.
- `-upgrade` Opt to upgrade modules and plugins as part of their respective installation steps. See the sections below for more details.

- 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.
- 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.
  - **-migrate-state:** will attempt to copy existing state to the new backend, and depending on what changed, may result in interactive prompts to confirm migration of workspace state. The **-force-copy** suppress these prompts and answer "yes" to the migration questions.
  - **-reconfigure:** disregards any existing configuration, preventing migration of any existing state.
  - **-backend=false:** Skip backend configuration.

- 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 will install the sources for any modules that were added to configuration since last init, but not change any already-installed modules. Use **-upgrade** to override this behavior, updating all modules to the latest available source code.



You can modify `terraform init`'s plugin behavior with the following options:

- `-upgrade` Upgrade all previously-selected plugins to the newest version that complies with the configuration's version constraints. This will cause Terraform to ignore any selections recorded in the dependency lock file, and to take the newest available version matching the configured version constraints.
- `-get-plugins=false` — Skip plugin installation.

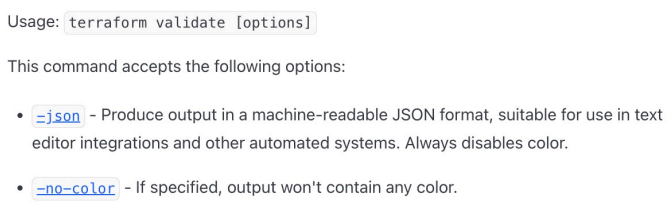
  Note: Since Terraform 0.13, this option has been superseded by the `provider_installation` and `plugin_cache_dir` settings. It should not be used in Terraform versions 0.13+, and this option was removed in Terraform 0.15.

- `-plugin-dir=PATH` — Force plugin installation to read plugins *only* from the specified directory, as if it had been configured as a `filesystem_mirror` in the CLI configuration. If you intend to routinely use a particular filesystem mirror then we recommend configuring Terraform's installation methods globally. You can use `-plugin-dir` as a one-time override for exceptional situations, such as if you are testing a local build of a provider plugin you are currently developing.
- `-lockfile=MODE` Set a dependency lockfile mode.

The valid values for the lockfile mode are as follows:

- readonly: suppress the lockfile changes, but verify checksums against the information already recorded. It conflicts with the `-upgrade` flag. If you update the lockfile with third-party dependency management tools, it would be useful to control when it changes explicitly.

- **Command: validate**
  - Validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider API, etc.
  - Validate runs check that verify whether a configuration is syntactically valid and internally consistent, regardless of any provider or existing state.
  - Validation requires an initialized working directory with any referenced plugins and modules installed.



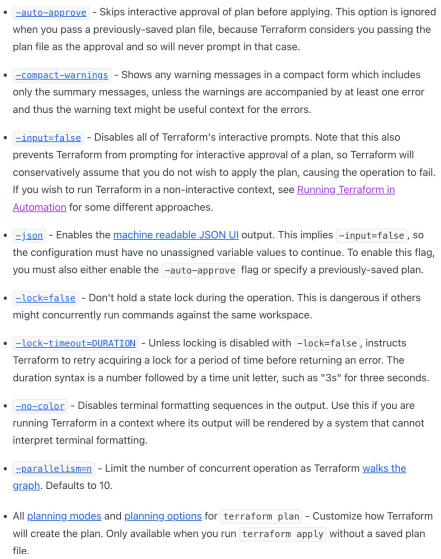
- **Command: plan**
  - Creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. By default, when Terraform creates a plan it:
    - Reads the current state of any already-existing remote objects to make sure that the Terraform state is up-to-date.
    - 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.
    - By default the **"apply"** command automatically generates a new plan and prompts for you to approve it.
    - You can use **-out=FILE** option to save the generated plan to a file on disk, which you can later execute by passing the file to terraform apply as an extra argument. This two-step workflow is primarily intended for when running Terrafomr in automation.
  - **Planning Modes**
    - **Destroy Mode**: creates a plan whose goal is to destroy all remote objects that currently exist, leaving an empty Terraform state. Active destroy mode using **-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 if Terraform. Activate refresh-only mode using **-refresh-only** command line option.
  - **Planning Options:** In addition to alternate planning modes, there are several options that can modify planning behavior. These options are available for both **terraform plan** and **terraform apply.**

- `-refresh=false` - Disables the default behavior of synchronizing the Terraform state with remote objects before checking for configuration changes. This can make the planning operation faster by reducing the number of remote API requests. However, setting `refresh=false` causes Terraform to ignore external changes, which could result in an incomplete or incorrect plan. You cannot use `refresh=false` in refresh-only planning mode because it would effectively disable the entirety of the planning operation.
- `-replace=ADDRESS` - Instructs Terraform to plan to replace the resource instance with the given address. This is helpful when one or more remote objects have become degraded, and you can use replacement objects with the same configuratation to align with immutable infrastructure patterns. Terraform will use a "replace" action if the specified resource would normally cause an "update" action or no action at all. Include this option multiple times to replace several objects at once. You cannot use `-replace` with the `-destroy` option, and it is only available from Terraform v0.15.2 onwards. For earlier versions, use `terraform taint` to achieve a similar result.
- `-target=ADDRESS` - Instructs Terraform to focus its planning efforts only on resource instances which match the given address and on any objects that those instances depend on.

  Note: Use `-target=ADDRESS` in exceptional circumstances only, such as recovering from mistakes or working around Terraform limitations. Refer to Resource Targeting for more details.
- `-var 'NAME=VALUE'` - Sets a value for a single input variable declared in the root module of the configuration. Use this option multiple times to set more than one variable. Refer to Input Variables on the Command Line for more information.
- `-var-file=FILENAME` - Sets values for potentially many input variables declared in the root module of the configuration, using definitions from a "tfvars" file. Use this option multiple times to include values from more than one file.

- **Command: apply**
  - The most straightforward way to use it is to pass it to run without any arguments at all, in which case it will automatically create a new execution plan and the prompt you to approve that plan, before taking the indicated actions.
  - Another way to use it is to pass it the filename of a saved plan file you created earlier with terraform plan -out=... in which case Terraform will apply the changes in the plan without any confirmation prompt. This two-step workflow is primarily intended for when running Terraform in automation.
  - **Usage:** the behavior of terraform apply differs significantly depending on whether you pass it the filename of a previously-saved plan file.
    - **Automatic Plan Mode:** The default case creates its own plan of action, in the same way terraform plan would.
      - Terraform will propose the plan to you and prompt you to approve it before taking the described actions, unless you waive that prompt by using the **-auto-approve** option.
      - When you run terraform apply default, you can use all of the planning modes and planning options available for terraform plan.
    - **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**; if you want to inspect a plan file before applying it, you use **terraform show.**
      - When using saved plan, **none** of the **planning** modes or **applying options** are **supported**.

- **Apply Options:**



- `-auto-approve` - Skips interactive approval of plan before applying. This option is ignored when you pass a previously-saved plan file, because Terraform considers you passing the plan file as the approval and so will never prompt in that case.
- `-compact-warnings` - Shows any warning messages in a compact form which includes only the summary messages, unless the warnings are accompanied by at least one error and thus the warning text might be useful context for the errors.
- `-input=false` - Disables all of Terraform's interactive prompts. Note that this also prevents Terraform from prompting for interactive approval of a plan, so Terraform will conservatively assume that you do not wish to apply the plan, causing the operation to fail. If you wish to run Terraform in a non-interactive context, see Running Terraform in Automation for some different approaches.
- `-json` - Enables the machine readable JSON UI output. This implies `-input=false`, so the configuration must have no unassigned variable values to continue. To enable this flag, you must also either enable the `-auto-approve` flag or specify a previously-saved plan.
- `-lock=false` - Don't hold a state lock during the operation. This is dangerous if others might concurrently run commands against the same workspace.
- `-lock-timeout=DURATION` - Unless locking is disabled with `-lock=false`, instructs Terraform to retry acquiring a lock for a period of time before returning an error. The duration syntax is a number followed by a time unit letter, such as "3s" for three seconds.
- `-no-color` - Disables terminal formatting sequences in the output. Use this if you are running Terraform in a context where its output will be rendered by a system that cannot interpret terminal formatting.
- `-parallelism=n` - Limit the number of concurrent operation as Terraform walks the graph. Defaults to 10.
- All planning modes and planning options for `terraform plan` - Customize how Terraform will create the plan. Only available when you run `terraform apply` without a saved plan file.

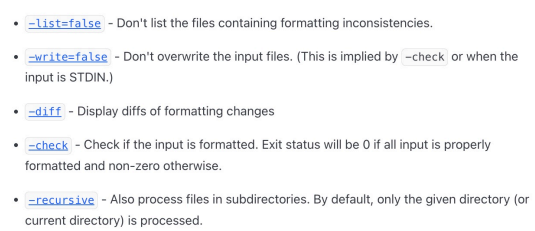
- **Command: destroy**
  - Is a convenient way to destroy all remote objects managed by a particular Terraform configuration.
  - **Usage:** This command is just a convenience alias for:

    terraform apply -destroy

    For that reason, this command accepts most of the options that terraform apply accepts, although it does not accepts a plan file argument and forces the selection of the "destroy" planning mode.
  - You can also create a **speculative destroy plan**, to see what the effect of destroying would be:

    terraform plan -destroy

- **Command: fmt**
  - Is used to rewrite Terraform configuration files to a canonical format and style. This command applies a subset of the Terraform language style conventions, along with other minor adjustments for readability.
  - **Usage:** By default **terraform fmt** scans the current directory for configuration files. If the dir argument is provided then it will scan that given directory instead. If dir is a single dash (-) the fmt will read from standard input.



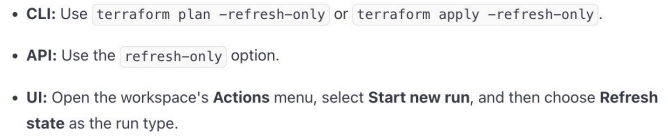
- `-list=false` - Don't list the files containing formatting inconsistencies.
- `-write=false` - Don't overwrite the input files. (This is implied by `-check` or when the input is STDIN.)
- `-diff` - Display diffs of formatting changes
- `-check` - Check if the input is formatted. Exit status will be 0 if all input is properly formatted and non-zero otherwise.
- `-recursive` - Also process files in subdirectories. By default, only the given directory (or current directory) is processed.

(7) Implement and maintain state

- **Backends**
  - A backend defines where Terraform's state snapshot are stored. A given Terrraform configuration can either specify a backend, integrate with Terraform Cloud, or do neither and default to storing state locally.
  - Backends primarily determine where Terraform stores its state. Terraform uses this persisted state data to keep of the resources it manages.
  - By default, Terraform implicitly uses a backend called local to store state as a local file on disk.
  - Some backends act like plain "remote disk" for state files; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies.
- **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.
  - Terraform has a **force-unlock** command to manually unlock the state if unlocking failed. Force unlock should be used to unlock your own lock in the situation where automatic unlocking failed. To protect you **force-unlock** command requires a unique ID. Terraform will output this lock ID if unlocking fails.
- **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.
  - **Usage:**

    terraform login [hostname]

  - By default Terraform will obtain an API token and save it in plain text in 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 the current configuration is not as desired.
- **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 objects. Applying **refresh-only run does not result in further changes to remote objects.**



- The **-refresh=false** option is used in normal planning mode to skip the default behavior of refreshing Terraform state before checking for configuration changes.

  - **CLI:** Use `terraform plan -refresh=false` or `terraform apply -refresh=false`.
  - **API:** Use the `refresh` option.

- **Replacing Selected Resources**
  - The replace option instructs Terraform to replace the object with the given resource address.

    - **CLI:** Use `terraform plan -replace=ADDRESS` or `terraform apply -replace=ADDRESS`.
    - **API:** Use the `replace-addrs` option.

- **Target Plan and Apply**
  - Resource Targeting is intended for exceptional circumstances only and should not be used routinely.

    - **CLI:** Use `terraform plan -target=ADDRESS` or `terraform apply -target=ADDRESS`.
    - **API:** Use the `target-addrs` option.

  - The usual caveats for targeting in local operations imply some additional limitations on Terraform Cloud features for remote plan created with targeting:
    - **Sentinel** policy checks for targeted plans will see only the selected subset of resources instances planned for changes in the tfplan import and tfplan/v2 import, which may cause an unintended failure for any policy that requires a planned change to a particular resource instance selected by its address.
    - **Cost Estimation** is disabled for any run created with -target set, to prevent producing a misleading understanding of cost due to resources instances being excluded from the plan.
    - You can **disable or constrain use of targeting** in a particular workspace **using Sentinel** policy based on the **tfrun.target addrs** value.

- **Generating Configuration**
  - When using import blocks to import existing resources, **Terraform can automatically generate configuration** during the plan for any imported resources that don't have an existing resource block. This option is enabled by default for runs started from UI or from VCS webhook.

    - **CLI:** Use `terraform plan -generate-config-out=generated.tf`.
    - **API:** Use the `allow-config-generation` option.

- **Terraform 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.

```
terraform {
  cloud {
    organization = "example_corp"
    ## Required for Terraform Enterprise; Defaults to app.terraform.io
    hostname = "app.terraform.io"

    workspaces {
      tags = ["app"]
    }
  }
}
```

- **Backend Configuration**
  - Each Terraform configuration can specify a backend, which defines where state snapshots are stored.
  - **You do not need to configure a backend when using Terraform Cloud** because Terraform Cloud automatically manages state in the workspaces associated with your configuration. If your configuration includes a **cloud block,** it cannot include a backend block.
  - **Backend Block:**

```
terraform {
  backend "remote" {
    organization = "example_corp"

    workspaces {
      name = "my-app-prod"
    }
  }
}
```

    - A configuration can **only provide one backend block**.
    - A backend block cannot refer to named values
    - If a configuration includes not backend block, Terraform **defaults to using local backend.**

  - **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, This lets you adopt backends without losing any existing state.
- To be extra careful, we always recommend manually backing up your state as well. You can do this by simply copying **terraform.tfstate** file to other location. The initialization process should create a backup as well, but it never hurts to be safe.

- **Partial Configuration**
  - You don't need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable if some arguments are provided automatically by an automation script running Terraform. Where 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.
  - There are several ways to supply the remaining arguments:
    - **File:** a configuration file may be specified via init command line. To specify a file, use the **-backend-config=PATH** option when running **terraform init.**
    - **Command-line key/value pairs:** key/value pairs can be specified via the init command line. To specify a single key/value pair, use the **-backend-config="KEY=VALUE"** option when running **terraform init**.
    - **Interactively**
  - The final, merged configuration is stored on disk in the **.terraform** directory, which should be ignored from version control. This means that sensitive information can be omitted from version control, but it will be present in plain text on local disk when running Terraform.
  - When using partial configuration, Terraform requires at minimum that an empty backend configuration is specified in one of the root Terraform configuration files.

```
terraform {
  backend "consul" {}
}
```

  - 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.

```
address = "demo.consul.io"
path    = "example_app/terraform_state"
scheme  = "https"
```

  - *.backendname.tfbackend is the recommended naming pattern.

- **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 this is what you want to do.
  - 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.

- **Sensitive Data in State**
  - Terraform state can contain sensitive data, depending on the resources in use and your definition of "sensitive". The state contains resources IDs and all resources attributes. For resources such a databases, this may contain initial passwords.
  - When using local state, state is stored in plain-text JSON files.
  - When using remote state, state is only held in memory when used by Terraform. It may be encrypted at rest, but this depends on the specific remote state backend.
    - **Terraform Cloud always encrypts state at rest and protects it with TLS** in transit. Terraform Cloud also knows the identity of the user requesting state and maintains a history of state changes. **Terraform Enterprise also supports detailed audit logging.**
    - Te **S3** backend supports **encryption at rest** when the encrypt option is enabled. **IAM policies** and logging can be used to identify any invalid access.

      Request for the state go over a **TLS** connection.

(8) Read, generate, and modify configuration

- Input variables let you customize aspects of Terraform modules without altering the module's own source code. This allows you to share module across different Terraform configurations, making your module composable and reusable.
- When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables. When you declare them in child modules, the calling module should pass values in the module block.
- If you're familiar with traditional programming languages, it can be useful to compare Terraform modules to function definitions:
  - Input variables are like function arguments.
  - Output values are like function return values
  - Local values are like a function's temporary local variables.
- Each input variable accepted by a module must be declared using a **variable** block.

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

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

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

  - The label after the variable keyword is a name for the variable, which must be unique among all variables in the same module. The name of a variable can be any valid identifier except the following: source, version, provider, count, for_each, lifecycle, depends_on, locals.
  - **Arguments:**

    - `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.

- Type constraints are created from a mixture of type keywords and type constructors. The supported type keywords are:
  - string
  - number
  - bool
- The type constructors allows you to specify complex types such as collections:
  - list(<TYPE>)
  - set(<TYPE>)
  - map(<TYPE>)
  - object({<ATTR NAME>= <TYPE>, ...}Martes es)
  - tuple([<TYPE>, ...])
- The keyword **any** may be used to indicate that any type is acceptable.
- If both **type** and **default** arguments are specified, the given default value must be convertible to the specified type.
- The description should concisely explain the purpose of the variable and what kind of value is expected.

○ **Custom Validation Rules**

- In addition to Type Constraints, a module author can specify arbitrary custom validation rules for a particular variable using a **validation** block nested within the corresponding **variable** 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) == "ami-"
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-
  }
}
```

- The condition argument is an expression that must use the value of the variable to return true if the value is valid, or false if it is invalid. **The expression can refer only to the variable that the condition applies to**, and must not produce errors.
- If the failure of an expression is the basis of the validation decision, use the **can** function to detect such errors.

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

  validation {
    # regex(...) fails if it cannot find a match
    condition     = can(regex("^ami-", var.image_id))
    error_message = "The image_id value must be a valid AMI id, starting with \"am
  }
}
```

- If condition evaluates to false, Terraform will produce an error message that includes the sentence given in **error_message.**
- Multiple validation blocks can be declared in which case **error message will be returned for all failed conditions.**

- **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 date will have access to the sensitive value in cleartext.

- **Using Input Variable Values**
  - Within the module that declared a variable, its value can be accessed from within expressions as var.<NAME>, where <NAME> matches the label given in the declaration block.

```
variable "user_information" {
  type = object({
    name    = string
    address = string
  })
  sensitive = true
}

resource "some_resource" "a" {
  name    = var.user_information.name
  address = var.user_information.address
}
```

  - Any expression whose result depends on the sensitive variable will be treated as sensitive themselves, and so in the above example the two arguments of resource "some_resource" "a" will also be hidden in the plan output.
  - A sensitive variable is a configuration-centered concept, and values are sent to providers without obfuscation. A provider error could disclose a value if that value is included in the error message.

- **Assigning Values to Root Module Variables**
  - When variables are declared in the root module of your configuration, they can be set in a number of ways:
    - In a Terraform Cloud Workspace
    - Individually, with the -var command line option

```
terraform apply -var="image_id=ami-abc123"
terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_ty
terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def4
```

    - In variable definitions (.tfvars) files, either specified on the command line or automatically loaded.

```
terraform apply -var-file="testing.tfvars"
```

- As environment variables.

```
$ export TF_VAR_image_id=ami-abc123
$ terraform plan
...
```

- If you provide values for undeclared variables defined **in a file** you will get a warning.
- If you provide values for undeclared variables defined on the **command line,** Terraform will error.

- **Variable Definition Precedence**
  - Terraform loads variables in the following order:
    - Environment Variables
    - The terraform.tfvars file, if present
    - The terrafomr.tfvars.json, if present
    - Any *.auto.tfvars or *.auto.tfvars.json file, if present
    - Any -var and -var-file options on the command line, in the order they are provided.

- **Output Variables**
  - Output values make information about your infrastructure available on the command line, and can expose information for other configurations to use.
  - Output values has several uses:
    - A child module can use outputs to expose a subset of its resource attributes to a parent module.
    - A root module can use outputs to print certain values in the CLI output after running **terraform apply**.
    - When using remote state, root module outputs can be accesses by other configurations via a terraform_remote_state data source.

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

    - The value argument takes an expression whose result is to be returned to the user.
    - Outputs are only rendered when Terraform applies your plan.

  - Accessing Child Modules Outputs

    module.<MODULE NAME>.<OUTPUT NAME>

- Optional Arguments
  - description
  - sensitive
  - depends_on: in less-common cases there are dependencies that cannot be recognized implicitly. This where you will use depends_on.

- **Vault Provider**
  - The vault provider allows Terraform to read from, write to, and configure HashiCorp Vault.

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

- **Data Sources**
  - Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.
  - A data source is accessed via a special kind of resource known as a data resource, declared using a **data** block.

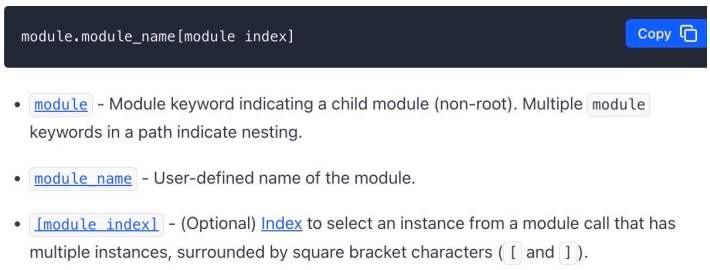
```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

  - A data block request that Terraform read from a given data source and export the result under the given local name. The name is used to refer to this resource from elsewhere in the same Terraform module, but has no significance outside the scope of a module.
  - Within the block body are query constraints defined by the data source. Most arguments depend on the data source.
  - The data resource cause Terraform only to read objects.
  - **Local-only Data Source**: 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.

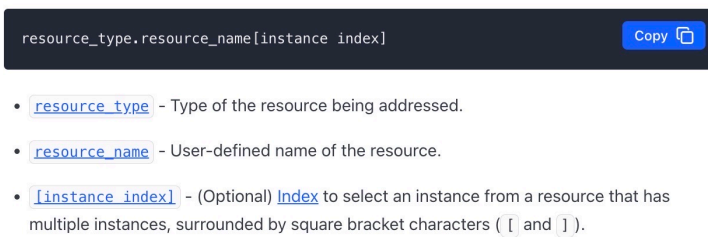
- **Resource Addressing**
  - A resource address is a string that identifies zero or more instances in your overall configuration. An address is made up of two parts:

    [module path][resource spec]

  - A module path addresses a module within the tree of modules:



  - A resource addresses a specific resource instance in the selected module



- Index values for Modules and Resources
  - **[N]:** where N is a 0-based numerical index into a resource with multiple instances specified by the count meta-argument.

  - **["INDEX"]:** where INDEX is a alphanumerical key index into a resource with multiple instances specified by the for_each meta-argument.

- **Built-in Functions**
  - Terraform language includes a number of built-in functions you can call from within expressions to transform and combine values.
  - Terraform language **does not support user-defined functions.**

- **Resource Graph**
  - Terraform builds a dependency graph from the Terraform configurations, and walks this graph to generate plans, refresh states, and more.
  - Graph Nodes:
    - Resource Nodes
    - Provider Configuration Node
    - Resource Meta-Node
  - When visualizing a configuration with **terraform graph,** you can see all of these nodes present.

- **Walking the Graph**
  - To walk the graph, a standard depth-first is done. Graph walking is done in parallel: a node is walked as soon as all of tis 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. By default up to 10 nodes in the graph will be processed concurrently. This number can be set using the **-parallelism** flag on the **plan, apply,** and **destroy** commands.

(9) Understand Terraform Cloud capabilities

- 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 providing changes to infrastructure, a private registry for sharing Terraform modules, detailed policy controls for governing the contents of Terraform configurations, and more.
- Terraform Cloud is available as a hosted service. Small teams can sign up for free to connect Terraform version control, share variables, run Terraform in a stable remote environment and securely store remote state. Paid editions allow you to add more that five users, create teams with different levels of permissions, and collaborate more effectively.
- Terraform Cloud Plus Edition allows organizations to enable audit logging, continuous validation, and automated configuration drift detection.
- Terraform Enterprise is a self-hosted distribution of Terraform Cloud. It offers enterprises a private instance that includes advanced features available in Terraform Cloud.

  - **Workspaces**
    - Terraform Cloud manages infrastructure collections with workspaces instead of directories. A workspace contains everything Terraform needs to manage a given collections of infrastructure, and separate workspaces function like completely separate working directories.
    - Terraform Cloud workspaces and local working directories serve the same purpose, but they store their data differently:

Component	Local Terraform	Terraform Cloud
Terraform configuration	On disk	In linked version control repository, or periodically uploaded via API/CLI
Variable values	As `.tfvars` files, as CLI arguments, or in shell environment	In workspace
State	On disk or in remote backend	In workspace
Credentials and secrets	In shell environment or entered at prompts	In workspace, stored as sensitive variables

        - In addition to the basic Terraform content, Terraform Cloud keeps some additional data for each workspace.
            - **state version**
            - **run history**

- **Teams:** are a group of Terraform Cloud users within an organization. The organization can grant workspace permissions to teams that allows its members to start Terraform runs, create workspaces variables, read and write state, etc.

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