### 

HashiCorp Terraform Standards Guide

Agnostic: Infrastructure

Date: January 30, 2019

Authors: morgantep@, thebo@, berlinsky@

Reviewers: sethvargo@, shikhman@, diekmann@

Contents

[**About this guide**](#_dpqpr2bfenwc) **4**

[**1. Style**](#_1vtoc6r30fun) **5**

[1.1 Terraform Modules](#_uy7ur7rag0v2) 5

[1.1.1 Module Structure](#_3uy0fkdg8ne) 5

[1.1.2 Naming Convention](#_38a7s511pak6) 5

[1.1.3 Variables](#_h1q5arnzwrv) 6

[1.1.4 Outputs](#_muk1m3pdc8wt) 6

[1.1.5 Data sources](#_dm9pffe4ohsl) 7

[1.1.6 Scripts (called by Terraform)](#_j07951chw04i) 7

[1.1.7 Helper scripts (not called by Terraform)](#_6kuycn10rzy2) 7

[1.1.8 Static files](#_oa7ec0hrccmo) 7

[1.1.9 Templates](#_4mnwubn6m8t) 7

[1.1.10 Resources](#_w2pgk6xrq29z) 8

[1.1.11 Formatting](#_pzfh2390zmxd) 8

[1.1.12 Expressions](#_1p9yz2sg9ohm) 8

[2.2 Common modules](#_lyp3j9gtxmxr) 9

[2.2.1 Structure](#_7wb2x3kj9tnl) 9

[2.2.2 Variables](#_g5cwbhjqegmv) 9

[2.2.3 Outputs](#_62q0veh0omih) 10

[2.2.4 Inline modules](#_cfctwpkeszgy) 10

[2.3 Root configs](#_hk4w0xwa5t6d) 10

[2.3.1 Directory Structure](#_6lpwkivmfan5) 11

[2.3.2 Outputs](#_x1urmmy0idag) 14

[2.4 Versioning](#_7h6ibcm01xp9) 14

[2.4.1 Terraform](#_q6o256c3mjr0) 14

[2.4.2 Google provider](#_avpx8t9xntlq) 14

[2.4.3 Modules](#_9a2av9gdacjj) 14

[2.4.4 Constrain by git reference](#_sziz3rfu5it1) 15

[2.4.5 Constrain by version](#_stayfnjwmnw9) 15

[**3. Cross-configuration communication**](#_875jc81689yk) **15**

[3.1 Remote state](#_tg38lvjo5eey) 15

[3.2 Data Sources](#_847ozrhsa55y) 16

[**4. Provisioning**](#_5wimls5cmzea) **16**

[4.1 Instances](#_lxdh0jbxuqrj) 16

[**5. Terraform Execution**](#_1se3sim3cvke) **17**

[5.1 Remote state](#_flz0rbhy1sar) 17

[5.2 Execution](#_fh3kvlbjssqw) 17

[5.3 Variable definitions](#_98fm6u1kca0x) 18

[**6. Version control**](#_ql7ticw42oe) **18**

[6.1 Branching strategy](#_vlz5ih6id8u3) 18

[6.2 Visibility](#_mtp6o8ereavl) 18

[6.3 Git best practices](#_4uplhr5hnpps) 19

[**7. Terraform operational best practices**](#_694yz7y7z6fb) **19**

[**8. Terraform security best practices**](#_i0zszhqhh1uf) **20**

# 

# About this guide

| **Document details** | |
| --- | --- |
| **Purpose** | This document serves as a guidepost for effective development with Terraform across multiple team members and workstreams. Our goal is to provide a unified and consistent approach to managing code by outlining some rules of engagement. These guidelines should be referenced during code reviews in order to align new development with existing work and to update old code.  As a disclaimer, the guidelines found here are not law but rather best practices that we have encountered in our numerous infrastructure-as-code experiences. In cases where new needs require deviation from this guide, we should adjust the guide. Please contribute by leaving feedback and suggestions in the comments section!  The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in [RFC 2119](https://www.ietf.org/rfc/rfc2119.txt). |
| **Intended audience** | These standards are targeted towards developers and administrators who will be writing Terraform code on a regular basis. |
| **Key**  **assumptions** | This guide assumes a basic familiarity with Terraform syntax/functionality and CI/CD best practices. |

# 

# 

# 1. Style

These standards cover basic style and structure for your Terraform configurations.

## 1.1 Terraform Modules

The following guidelines apply to [reusable Terraform modules](#chv93ewt58zr) and [root configs](#i04g9yg61s0f).

### 1.1.1 Module Structure

1. Terraform modules must follow the [standard module structure](https://www.terraform.io/docs/modules/create.html#standard-module-structure).
2. Every module must start with a main.tf file, where resources are located by default.
   * Among other resources, all [local](https://www.terraform.io/docs/configuration/locals.html)s should be defined here in a single locals block.
3. All modules must have a README.md (with basic documentation in Markdown format).
4. Examples should be located in examples/, each with its own subdirectory and a README.md file.
5. Logical groupings of resources can be grouped into their own files and given descriptive names, like network.tf, instances.tf, or loadbalancer.tf.
   * Avoid giving every resource its own file. Resources should be grouped by shared purpose. For example, “google\_dns\_managed\_zone” and “google\_dns\_record\_set” would likely be combined in dns.tf.
6. Only Terraform (\*.tf) and repo metadata files (like README.md, CHANGELOG.md, or kitchen.yml) should exist at the root directory of a module.
7. Additional documentation should be stored in a docs/ subdirectory.

### 1.1.2 Naming Convention

All configuration objects should be named using underscores to delimit multiple words. This practice ensures consistency with the naming convention for resource types, data source types, and other predefined values. Note that this convention does not apply to name arguments.

| # Good  resource "google\_compute\_instance" "web\_server" {  name = “web-server”  # ... }  # Bad  resource “google\_compute\_instance” “web-server” {  name = “web-server”  # …  } |
| --- |

### 1.1.3 Variables

1. All variables shall be declared in variables.tf.
2. Variables shall have descriptive names relevant to their usage or purpose.
   1. Inputs, local variables, and outputs representing numeric values such as disk sizes or RAM size SHOULD be named with units (like ram\_size\_gb). GCP APIs do not have standard units, so naming variables with units makes the expected input unit clear for configuration maintainers.
   2. For units of storage, the unit prefix (kilo, mega, giga) SHOULD be binary (powers of 1024). For all other units of measurement, the unit prefix SHOULD be decimal (powers of 1000). This matches the usage within GCP.
   3. Boolean variables SHOULD be named with positive values (like enable\_external\_access) to simplify conditional logic.
3. Variables must have descriptions. These are automatically included in any published modules’ auto-generated documentation [through terraform-docs](https://github.com/segmentio/terraform-docs). Descriptions add additional context for new developers that descriptive names cannot.
4. Variables should have defined types.
5. Variables with non-environment-specific values (like disk size) should be given default values.
6. Variables for environment-specific values (like project\_id) should not be given defaults. This forces the calling module to provide meaningful values.
7. Variables should only have empty defaults (like empty strings or lists) where leaving the variable empty is a valid preference which will not be rejected by the underlying API(s).
8. Be thoughtful in your use of static literals (hardcoded strings, etc.) and parameterize anything which must vary per instance or environment. [Local values](https://www.terraform.io/docs/configuration/locals.html) can be used in cases where a literal is reused in multiple places without exposing it as a variable.
9. When deciding whether to expose a variable, ensure that you have a concrete use case for changing that variable. Don’t expose variables on the off chance that it’s needed.
   1. Adding a variable with a default value is backwards compatible, and thus “cheap.”
   2. Removing a variable is backwards incompatible, and thus “expensive.”
10. Boolean variables should be explicitly declared as strings until Terraform adds further support for [boolean input variables](https://www.terraform.io/docs/configuration/variables.html#booleans).

### 1.1.4 Outputs

1. All outputs shall be organized into outputs.tf.
2. Outputs should have meaningful descriptions.
3. Output descriptions should be documented in the README. Descriptions should also be auto-generated on commit [with terraform-docs](https://github.com/segmentio/terraform-docs).
4. Make an effort to output all the useful values root modules would want to reference or share with modules. Particularly for open source or heavily used modules, expose all outputs that have potential for consumption.

### 1.1.5 Data sources

1. Data sources are located adjacent to the resources which reference them.
   1. If you are fetching an image to be used in launching an instance, you can place it alongside the instance instead of collecting data resources in their own file.
2. If the number of data sources grows considerably, it’s a reasonable practice to move these to a dedicated data.tf file.
3. Data sources should use variable or resource interpolation, where appropriate, to fetch data relative to your current environment.

### 1.1.6 Scripts (called by Terraform)

1. Bespoke scripts can be called by Terraform through provisioners, including the local-exec provisioner.
2. Custom scripts should be avoided, if possible, and constrained to instances where native Terraform resources do not support the desired behavior. Any custom scripts used must have a clearly documented reasoning and ideally a deprecation plan. Additionally, custom scripts should not replace configuration management tools in cases where they are more appropriate.
3. Bespoke scripts called by Terraform must be organized into scripts/.
4. Use scripts only when absolutely necessary, as the state of resources created through scripts is not accounted for or managed by Terraform. You’ll likely want to add a policy of ignore\_changes = [\*] on such resources.

### 1.1.7 Helper scripts (not called by Terraform)

1. Helper scripts should be organized in a ./helpers directory.
2. Helper scripts shall be documented in the README with an explanation and example invocations.
3. Helper scripts accepting arguments should provide argument-checking and --help output.

### 1.1.8 Static files

1. Static files which are referenced by Terraform (like Startup scripts loaded onto GCE instances) but not executed must be organized into files/.
2. Lengthy heredocs should be externalized from their HCL and into external files. These should be referenced with the [file() function](https://www.terraform.io/docs/configuration/interpolation.html#file-path-).

### 1.1.9 Templates

1. Files which are injected with the Terraform [template\_file resource](https://www.terraform.io/docs/providers/template/d/file.html) should be given the file extension .tpl.
2. Templates must be placed in templates/.

### 1.1.10 Resources

1. Resources that are the only one of their type (i.e., a single load balancer for an entire module) should be named ‘main’ to simplify references to that resource.
   * It takes extra mental work to remember some\_google\_resource.my\_unique\_name.id vs. some\_google\_resource.main.id.
2. Resources that share the same type as others in the same module should be given meaningful names to differentiate them.
3. Resources must be named in snake-case (like db\_instance).
4. Resource names should be singular.
5. Resource names shouldn’t repeat the resource type within the name. For example:  
   Do this: resource "google\_compute\_global\_address" "main" { ... }  
   Not this: resource "google\_compute\_global\_address" "main\_global\_address" { … }
6. Ensure that [deletion protection](https://www.terraform.io/docs/configuration/resources.html#prevent_destroy) is enabled for stateful resources like databases. For example:

| resource "google\_sql\_database\_instance" "main" {  name = "master-instance"  settings {  tier = "D0"  }   **lifecycle {  prevent\_destroy = true  }** } |
| --- |

### 1.1.11 Formatting

1. All Terraform files must conform to the standards of terraform fmt.

### 1.1.12 Expressions

1. Limit the complexity of any individual interpolated expressions. If many functions are needed in a single expression, consider splitting it out into multiple expressions using [locals](https://www.terraform.io/docs/configuration/locals.html).
2. Never have more than one ternary operation in a single line. Instead, use multiple local values to build up the logic.
3. Be sparing when using user-specified variables to set the count variable for resources.
   * If a resource attribute is provided for such a variable (like project\_id) and that resource does not yet exist, Terraform will not be able to generate a plan and will report the error “[value of count cannot be computed](https://github.com/hashicorp/terraform/issues/17421).”
4. Use count to instantiate a resource conditionally. For example:

| variable "readers" {  description = "..."  type = "list"  default = []  }  resource "foo" "bar" {  // Do not create this resource if the list of readers is empty.  count = "${length(var.readers) == 0 ? 0 : 1}"  ...  } |
| --- |

## 

## 2.2 Common modules

Modules that are meant for reuse should follow the following standards, as well as the normal [Terraform guidelines](#_uy7ur7rag0v2).

### 2.2.1 Structure

1. All common modules should have an [OWNERS](https://github.com/bkeepers/OWNERS) file (or [CODEOWNERS](https://blog.github.com/2017-07-06-introducing-code-owners/) on GitHub) documenting who is responsible for the module.
2. Common modules should follow [SemVer v2.0.0](https://semver.org/spec/v2.0.0.html) when new versions are tagged/released.
3. Modules must not declare providers or backends. Leave that to the root modules.
   * Working examples should codify if a specific provider version is needed for a given module.

### 2.2.2 Variables

It’s a good practice to allow flexibility in the labelling of resources through the module’s interface. Consider providing a labels variable with a default value of an empty map to apply throughout labelable resources:

| variable "labels" {  description = "A map of labels to apply to contained resources."  default = {}  type = "map"  } |
| --- |

### 2.2.3 Outputs

Outputs are required for common modules that define resources.

* 1. Variables and outputs are used to infer dependencies between modules and resources. Without any outputs, users cannot properly order your module in relation to their Terraform configurations.
  2. Every resource defined in a common module should have at least one output which references that resource.

### 2.2.4 Inline modules

1. Inline modules may be used to organize complex Terraform modules into smaller units, or de-duplicate common resources.
2. Inline modules shall be placed in modules/$modulename.
3. Inline modules should be treated as private and should not be used by outside modules, unless the common module specifically documents them otherwise.
4. Be aware that Terraform doesn’t track refactored resources; if you start out with a number of resources in the top level module and then push them into submodules, Terraform will try to recreate all refactored resources.
5. Outputs defined by internal modules are not automatically exposed; if you want to share outputs from internal modules you’ll need to re-output them.

## 

## 2.3 Root configs

Root configs, or root modules, are the working directories from which you run the Terraform CLI. They should follow the following standards, as well as the normal [Terraform guidelines](#_uy7ur7rag0v2) where applicable. Explicit recommendations for root modules supersede the general guidelines.

Resources for different applications and projects should be separated into their own Terraform directories that *can* be managed independently of each other. A service might represent a particular application or a common service like shared networking. Importantly, all the Terraform code for a particular service should be nested under **one** directory (including subdirectories).

### 2.3.1 Directory Structure

There are multiple ways to organize Terraform root configurations, especially when it comes to managing multiple environments. When it comes to managing the Terraform config for a particular service, the recommended structure is to use environment directories.

#### Directories per environment

In this style, each service must split its Terraform config into multiple directories. In this structure, the directory layout must be as follows:

| -- SERVICE-DIRECTORY/  -- OWNERS  -- modules/  -- service/  -- main.tf  -- variables.tf  -- outputs.tf  -- README  -- ...other…  -- environments/  -- dev/  -- backend.tf  -- main.tf  -- provider.tf  -- qa/  -- backend.tf  -- main.tf  -- provider.tf  -- prod/  -- backend.tf  -- main.tf  -- provider.tf |
| --- |

##### Environment directories

Each environment directory within corresponds to a [Terraform Workspace](https://www.terraform.io/docs/state/workspaces.html) and deploys a version of the service to that environment. This config should reference modules to share code across environments, including typically a service module which includes the base shared Terraform config for the service.

This environment directory must contain the following files:

* A backend.tf file declaring the Terraform [backend](https://www.terraform.io/docs/backends/) state location (typically GCS).
* A main.tf file which instantiates the service module.
* A provider.tf file which declares provider configuration.

#### Workspaces per environment

Alternatively, a single Terraform directory can be used per service and shared across environments. Each environment would have its own [workspace](https://www.terraform.io/docs/state/workspaces.html).

When using workspaces, all environments share the same modules, and the configuration is driven by a tfvars file and a workspace. Workspaces are helpful in that they limit the amount of code that must be copy-pasted between environment directories, which can help enforce parity between environments while maintaining their own state files.

By default a single workspace named “**default”** exists. It is recommended to create and use a workspace for each environment and use the **“default”** workspace only when working with resources that may be used across multiple environments like some service accounts.

Workspaces can be listed with the workspace subcommand:

| > terraform workspace list  \* default |
| --- |

To create a new workspace:

| > terraform workspace new prod  Created and switched to workspace "prod"!  You're now on a new, empty workspace. Workspaces isolate their state,  so if you run "terraform plan" Terraform will not see any existing state  for this configuration. |
| --- |

Workspaces isolate their state by creating an additional state file in the state backend for each workspace.

You must select a workspace before issuing terraform commands to that workspace.

| > terraform workspace select prod  Switched to workspace "prod". |
| --- |

Once you have switched to a workspace, all terraform commands work in the same manner you are accustomed to. A tfvars should be kept for each workspace to encapsulate the inputs for a given environment. tfvars can be stored in GCS or encrypted and stored alongside code in a source code management system using KMS or a tool like *git-crypt* or *git-secret*.

| > terraform apply -var-file=prod.tfvars |
| --- |

The tfvars file will drive the configuration of a specific environment. To enable/disable resources per environment, it is common to use the count attribute.

Example: prod.tfvars

| replica\_count = 5 |
| --- |

Example: staging.tfvars

| replica\_count = 1 |
| --- |

Example: dev.tfvars

| replica\_count = 0 |
| --- |

Example: main.tf

| resource "google\_sql\_database\_instance" "master" {  name = "${terraform.workspace}-master"  //...  }  resource "google\_sql\_database\_instance" "replica" {  name = "${terraform.workspace}-replica-${count.index}"  count = "${var.replica\_count}  //...  } |
| --- |

### 2.3.2 Outputs

1. Information from a root module which other root modules may depend on must be exported as outputs.
2. Root module outputs can be referenced using remote state.

#### Publishing outputs with remote states

1. Make sure to re-output nested module outputs that are useful as remote state.
   * Only root module-level outputs can be referenced from other Terraform environments/applications.
2. Information related to a service’s endpoints should be exported to remote state to allow use by other dependent apps for configuration.

## 

## 2.4 Versioning

### 2.4.1 Terraform

Terraform v0.12 is a significant release that will include some backwards incompatibilities. Pin the Terraform version (example given below) to a known safe version until v0.12 has been released and stabilized.

| terraform {  required\_version = "~> 0.11.10"  } |
| --- |

### 2.4.2 Google provider

Pin the Google provider to a known good version, and make updating the version pin a regular practice.

| provider "google" {  version = "~> 1.19.1"  } |
| --- |

### 2.4.3 Modules

References to shared modules must be [constrained](https://www.terraform.io/docs/modules/usage.html#module-versions) to a release tag. Targeting a specific commit hash or branch is dangerous as it gives no context to the version of the underlying module. Updating modules should involve as little guesswork as possible for both authors and reviewers.

### 2.4.4 Constrain by git reference

References to shared modules may be constrained to any arbitrary git reference (commit, branch, or tag). For reasons outlined above, we only recommend using this to reference tags:

| module "vpc" {  source = “git::https://github.com/terraform-google-modules/terraform-google-network?ref=v0.4.0”  ...  } |
| --- |

### 2.4.5 Constrain by version

When a git tag is released to the Terraform Module Registry, it creates a numbered version of that module (note that this does not apply to Github repositories, only to modules released to registries). An invocation can be constrained to said version:

| module "nat\_gateway" {  source = “GoogleCloudPlatform/nat-gateway/google”  version = “1.2.2”  ...  } |
| --- |

# 

# 3. Cross-configuration communication

A common problem that arises when using Terraform is how to share information across different Terraform configurations (possibly maintained by different teams). Two strategies for doing so are outlined below.

Generally, information can be shared between configurations without requiring they be stored in a single configuration directory (or even a single repo).

## 3.1 Remote state

1. Remote state should be used to publish information between different Terraform configurations. In addition to the runtime benefits of remote state (such as locking), remote state provides a uniform way of sharing information between different Terraform configurations.
2. Remote state must be used instead of local state. [GCS](https://www.terraform.io/docs/backends/types/gcs.html) or [Terraform Enterprise](https://www.terraform.io/docs/backends/types/terraform-enterprise.html) are preferred state backends.
3. Remote state should be queried using the [remote\_state](https://www.terraform.io/docs/providers/terraform/d/remote_state.html) data source.

## 

## 3.2 Data Sources

1. GCP data sources should be used for querying resources that are not managed by Terraform.
2. GCP data sources should be used for querying resources that are implicitly created or managed by Terraform or GCP, such as the [default GCE service account](https://www.terraform.io/docs/providers/google/d/google_compute_default_service_account.html).
3. GCP data sources should not be used for querying resources that are managed by another Terraform configuration. Using data sources to query outside resources adds implicit dependencies on resource names and structures that may be unintentionally broken by normal Terraform operations. Prefer publishing information with [remote states](#16534on3a6i) and the [remote\_state](https://www.terraform.io/docs/providers/terraform/d/remote_state.html) data source.

# 

# 4. Provisioning

Terraform can be used to provision many forms of cloud infrastructure, including virtual machines, which have their own set of considerations.

## 4.1 Instances

1. When configuration management is used to configure VMs, Terraform should hand off to the configuration management tool with a provisioner block
2. Terraform should provide VM configuration information to configuration management with instance metadata.

Resources that require teardown logic should use provisioner with when = destroy to clean up old state associated with the instance.

# 

# 

# 5. Terraform Execution

Keeping your infrastructure secure depends on having a stable and secure process for applying Terraform updates.

## 5.1 Remote state

1. Terraform should use remote state to prevent race conditions and stale Terraform plans from corrupting/mismanaging resources.
2. Terraform should use remote state to communicate configuration between different Terraform root modules.
3. The bucket used for remote state should only be accessible by the build system or highly privileged administrators.
4. The GCS backend allows passing in a [Customer-Supplied Encryption Key](https://cloud.google.com/security/encryption-at-rest/customer-supplied-encryption-keys/) (CSEK) to encrypt state client-side. CSEKs may be used for encryption of Terraform state at rest.

## 5.2 Execution

1. Terraform should execute via automated tooling, to ensure consistent execution context.
   1. Where available, Terraform Enterprise should be used to run terraform plan and terraform apply.
   2. If a build system (like Jenkins or Cloud Build) is already in use and Terraform Enterprise is not, the build system may be used to run terraform plan and terraform apply.
2. Terraform executions should always generate a plan first (and [saved to an output file](https://www.terraform.io/guides/running-terraform-in-automation.html)), and then the plan should be executed once approved by an infrastructure owner.
3. **Decide if Terraform is a service or a user.** Terraform is similar to tools like git in that it may be run as a user (Alice committed this code) or as a machine (this CI/CD server checked out this code). Executing Terraform from a machine in a CI/CD pipeline is the recommended approach.
   1. When Terraform is being run as a user, you have two options:
      1. Use a service account key. Users can download service account keys from the Google Cloud Console web UI or use the gcloud CLI tool. In this case, Terraform would run as the service account and inherit its IAM roles.
      2. Authenticate via user credentials by running gcloud auth application-default login to generate application default credentials. This method can be used, but is not recommended due to relying on gcloud tokens generated in a hidden project and can cause issues with API activation.
   2. When Terraform is being run as a machine on GCP, authenticate using the GCE/GKE service account. This is the recommended approach. When Terraform is being run as a machine on another provider, use a dedicated service account.

## 5.3 Variable definitions

1. Variables should only be provided with a .tfvars variables file for root modules.
   1. Variable files should be named terraform.tfvars for consistency.
2. Variables must not be specified with the -var ‘key=val’ option, or with environment variables.
   1. CLI options and environment variables are ephemeral and easy to forget; using a variables file is much more predictable.
   2. An exception may be made when providing sensitive information within a system like Jenkins or Terraform Enterprise, in which case providing such sensitive variables as [environment variables](https://www.terraform.io/docs/configuration/environment-variables.html#tf_var_name) is preferred.
      1. Keep in mind that such sensitive values will still end up in the state file and may also be exposed as outputs.

# 

# 6. Version control

As with other forms of code, infrastructure code should be stored in version control to preserve history and allow easy rollbacks.

## 6.1 Branching strategy

This branching strategy applies to all repositories which contain Terraform code.

1. The master branch is the primary development branch and represents the latest approved code. The master branch is [protected](https://docs.gitlab.com/ee/user/project/protected_branches.html).
2. Development happens on feature and bug fix branches, which branch off master.
   * Feature branches should be named feature/$feature\_name.
   * Bug fixes should be named bugfix/$bugfix\_name.
3. When a feature or bug fix is complete, it should be merged back into master using a pull request.
4. Branches should be rebased before being merged, to prevent merge conflicts.

## 

## 6.2 Visibility

1. Terraform source code should be visible to both infrastructure owners (e.g., SREs) and infrastructure stakeholders (e.g., developers).
   1. The intent of this visibility is to ensure that infrastructure stakeholders can have a better understanding of the infrastructure that they depend on.
2. Terraform source code should only be executable by infrastructure owners and/or build system to ensure that all changes are done by a user that can react to unexpected changes.
3. Infrastructure stakeholders should be encouraged to submit merge requests as part of the change request process.

## 

## 6.3 Git best practices

1. Use version control and treat Terraform configurations as versionable code artifacts.
2. Never commit secrets to source control.
3. Proper git development flow is encouraged, including feature branches, code reviews, and pull requests leading to merges with the primary branch.
4. If a module has an OWNERS file, one of the owners should approve a pull request before it is merged.

# 

# 7. Terraform operational best practices

In general, a number of principles should be followed when operating Terraform, whether directly on user workstations or through an automation pipeline.

1. Always run terraform plan before terraform apply, and ideally, use the plan’s output as the input to the apply operation to guarantee a changeset. Review the resources which will be added, modified, and destroyed, and make sure they are as you’d expect.
2. Avoid terraform import-ing existing resources, instead preferring to create new resources and sunset the old ones when appropriate.
   1. terraform import should only be used with explicit approval, and only in cases where sunsetting the old resource would require significant toil.
   2. If terraform import is to be used, you should first program the new resource as though it would be launched from scratch, and then personally verify that all parameters match the live instance, before importing it.
3. When developing new infrastructure or debugging problems through the GCP console or gcloud, reflect your solution in code, undo any changes you made manually, and test the solution by re-applying your Terraform code.
4. Use separate root modules to target different logical environments instead of workspaces or separate variable files.
   1. Prefer separate root modules (described in the [Root Configs](#i04g9yg61s0f) section) for long-lived environments that provide dependencies to other resources.
   2. Workspaces may be used for short lived/ephemeral resources, but should be limited in scope to small groups of resources that do not provide dependencies to other resources.
5. Never modify the Terraform state by hand
   1. Use [terraform state](https://www.terraform.io/docs/commands/state/index.html) CLI instead.
   2. When using remote state, use [terraform state pull](https://www.terraform.io/docs/commands/state/pull.html) modify the state with [terraform state](https://www.terraform.io/docs/commands/state/index.html), and push the resulting state with [terraform state push](https://www.terraform.io/docs/commands/state/push.html).
   3. Ensure that Terraform state files are [gitignored](https://github.com/github/gitignore/blob/master/Terraform.gitignore) to prevent accidentally committing them to source control.
6. Regularly review version pins for Terraform, Terraform providers, and modules. Pinning versions ensures stability but prevents bug fixes and other improvements from being incorporated into your configuration.
7. You can create aliases to make local development easier by adding these to your BASH profile:
   1. alias tf="terraform"
   2. alias terrafrom="terraform"

# 

# 8. Terraform security best practices

Terraform requires sensitive access to your cloud infrastructure to operate. Following these security best practices can help to minimize the associated risks and improve your overall cloud security posture.

1. **Use remote state.** For GCP customers, using the [GCS State Backend](https://www.terraform.io/docs/backends/types/gcs.html) is highly recommended. Not only does this provide the ability to lock the state to allow for collaboration as a team, it also separates the state and all the potentially sensitive information from VCS.
2. **Avoid storing secrets in state.** There are many resources and data providers in Terraform that store secret values in plain text in the state file. It is best to avoid storing secrets in state, though at times automation may be sacrificed. Below are some examples of providers that store secrets in plain text:
   1. [vault\_generic\_secret](https://www.terraform.io/docs/providers/vault/r/generic_secret.html)
   2. [tls\_private\_key](https://www.terraform.io/docs/providers/tls/r/private_key.html)
   3. [google\_service\_account\_key](https://www.terraform.io/docs/providers/google/r/google_service_account_key.html)
   4. [datasource\_client\_config](https://www.terraform.io/docs/providers/google/d/datasource_client_config.html)
3. **Encrypt sensitive values.** Certain Terraform resources have the **pgp\_key** attribute, which allows those values to be encrypted with a known PGP public key. A practical use case would be allowing developers to encrypt a value with the public key of the build system. While this is incredibly useful and should be used wherever it is not possible to avoid managing secrets in Terraform, this is only present on certain resources.
4. **Encrypt the state.** The GCS backend in Terraform allows for passing in [CSEKs at runtime](https://www.terraform.io/docs/backends/types/gcs.html) using the **GOOGLE\_ENCRYPTION\_KEY** environment variable. Though GCS buckets are encrypted at rest, this gives an added layer of protection. Even though there should not be secrets in the state file, you should always encrypt the state as a measure of defense in depth. As a side note, using a CMEK does not require extra IAM permissions, to access the state file, since only server side encryption is supported.
5. **Modularize where possible.** Terraform gives the ability to pass variables in at run time, which, where possible, should be used for a few reasons:
   1. Modularizing reduces repetitive code, which can lead to configuration drift and errors over time.
   2. Variable injection at runtime enables and encourages unit testing of Terraform code as part of a CI/CD pipeline.
6. **Execute Terraform programmatically.** From a security perspective, Terraform is a highly sensitive product having vast control over your infrastructure. Much like deploying applications should be scoped to systems like Cloud Build and Spinnaker, infrastructure deployments with Terraform should be run programmatically as part of a pipeline, when possible, using [service accounts](https://cloud.google.com/iam/docs/service-accounts) instead of Cloud Identity users.
7. **Separation of duties.** If it is not possible to run Terraform from an automated system where no users have access, at the very least, separate permissions and directories in such a way that a separation of duties is adhered to. For example, a network project would correspond with a network Terraform service account or user whose access is limited to this project.
8. **Run pre-apply checks.** When running Terraform in an automated pipeline, use [Google’s Terraform Validator](https://github.com/GoogleCloudPlatform/terraform-validator) to check the plan output against existing [Forseti](https://forsetisecurity.org/) policies to ensure that an apply action will not cause security regressions.
9. **Run post-apply checks.** Once the Terraform apply has executed, ensure that some type of integration checks are run from a security perspective in an automated manner. Tools like [Forseti](https://forsetisecurity.org/), [Inspec](https://inspec.io) and [Serverspec](https://serverspec.org/) are all valid choices for this type of check.