Terraform Cloud Tooling

Infrastructure as Code (IaC) and CI/CD

Date: August 10, 2022

Contents

[**1.0 Overview**](#_5az8bssoho5a) **2**

[**2.0 Design Principles**](#_ncwg5p296edh) **2**

[**3.0 Infrastructure as Code (IaC)**](#_559l8y7vpd0g) **2**

[3.1 Terraform](#_j49hsz77izqa) 3

[3.1.1 Module Scaffolding](#_aearnbz2z6x8) 3

[3.2 Terragrunt](#_7dlmcevtrl92) 4

[3.2.1 Keeping Terraform code DRY](#_nf1rhshl72c8) 4

[3.2.2 Keeping remote state configuration DRY](#_xe1sghkx5jr6) 4

[3.2.3 Executing Terraform on multiple modules at once](#_esb60buge4cy) 5

[**4.0 Continuous Integration / Continuous Deployment (CI/CD)**](#_j9ydp66e3qsb) **5**

[4.1 Code Repository](#_95dnhaljzrgb) 5

[4.2 Development Branching Model (Trunk vs Gitflow)](#_dx7txdgzm9mv) 5

[4.3 CI/CD Pipelines](#_btb25w893w4) 7

## 1.0 Overview

This document will describe the solution design for achieving automated deployment and management of Azure infrastructure through the use of infrastructure-as-code (IaC) and CI/CD tooling.

## 2.0 Design Principles

The solution adheres to the following design principles:

1. All Azure infrastructure will be managed as code
2. All infrastructure code will reside in a code repository
3. Changes to live configuration code will use a trunk based branching model, while changes to Terraform module code will use a Gitflow branching model
4. Changes to code will require peer review before being merged
5. Changes to code will trigger CI/CD pipelines to deploy into the live environments

## 3.0 Infrastructure as Code (IaC)

All Azure infrastructure will be managed as code, using a combination of Terraform and Terragrunt.

### 3.1 Terraform

Terraform codifies cloud APIs and provides a common declarative configuration syntax (HCL) which DevOps teams can use to describe infrastructure resources and their attributes as code.

When Terraform is ran, it builds a dependency graph[[1]](#footnote-1) of the resources described in the configuration files and applies the configuration in an idempotent manner (ie. it only applies the changes needed to align the infrastructure state with the declared desired state). In order to this, Terraform makes use of state files[[2]](#footnote-2) to map infrastructure resources to their configuration.

To promote reusability, the Terraform language includes modules[[3]](#footnote-3) which are containers for multiple resources that are used together. The main Terraform configuration (aka ‘root’ module) can invoke child modules multiple times and supply each invocation with different inputs.

#### 3.1.1 Module Scaffolding

In any scenario where we are undertaking the effort of creating and maintaining a Terraform module due to a lack of a suitable pre-existing one in the community, we will work from a common starting point by utilizing a Terraform module generator:

<https://github.com/sudokar/generator-tf-module>

This module generator provides scaffolding / boilerplate code for the following:

* main.tf, variables.tf and outputs.tf files
* .editorconfig file to help maintain consistent coding styles for multiple developers across various [EditorConfig](https://editorconfig.org/) compatible editors and IDEs such as Visual Studio Code
* .terraform-version file for specifying the module’s minimum required version of Terraform. This is leveraged by tools such as [tfenv](https://github.com/tfutils/tfenv) to automatically switch between multiple installed versions of Terraform
* example directory with module usage tf files, to demonstrate how to use the module
* test directory with example test based on chosen framework (in our case, terratest)
* .pre-commit-config.yaml file with predefined hooks to be run prior to every commit, including terraform fmt, terraform-docs, check-merge-conflict, go fmt and golint
* .gitignore and .gitattributes files

#### 3.1.2 Testing Framework

As noted in the module scaffolding section above, we recommend using Terratest as a testing framework for the Terraform modules.

Terratest is a Go library which provides patterns and helper functions for testing infrastructure. It executes IaC tools (in our case, Terraform) to deploy infrastructure in a real environment, runs your tests against the live infrastructure to validate that it works correctly, and then destroys the infrastructure at the end of the test.

The benefits of Terratest and Test Driven Development (TDD) more generally, are many, including:

* Tests whether the module / infrastructure actually works
* Requirements driven, with acceptance criteria defined ahead of time
* Results in optimized code, since the only code written is that which is needed to get the test to pass
* Gives engineers early feedback which makes it easier to identify issues, shortening development time
* Increases code test coverage, which helps identify regressions in new code
* It becomes easier to refactor code
* It becomes easier to add and test new code / features down the road
* Code becomes living / up-to-date documentation

### 3.2 Terragrunt

As environments and the infrastructure within them grow in size and complexity, operational challenges begin to present themselves. With Terraform, there are three main challenges developers typically face which (among others) are addressed by Terragrunt:

1. Keeping Terraform code DRY
2. Keeping remote state configuration DRY
3. Executing Terraform on multiple modules at once

#### 3.2.1 Keeping Terraform code DRY

One of the advantages of managing infrastructure as code is repeatability. It becomes trivial to provision and manage multiple instances of resources, and is a necessity for many different use cases such as having ‘copies’ of infrastructure in different environments. However, we don’t want to repeat the same resource declarations or module invocations multiple times, when only the values of the resource’s attributes are different.

Terragrunt not only lets you separate your Terraform modules from the live configuration of the infrastructure, but doesn’t actually require any Terraform configuration in the folder structure at all. Instead, there is just a terragrunt.hcl file which includes a terraform {...} block that specifies the location of the Terraform module to be used, and an inputs {...} block where you supply the environment-specific values for the module’s input variables.

#### 3.2.2 Keeping remote state configuration DRY

As noted in the section on Terraform, it uses state files to map infrastructure resources to their configuration. Due to how Terraform processes the configuration and builds a resource dependency graph, Terraform does not support variable interpolation, expressions or functions in the remote state configuration.

If we are following best practices and have our Terraform code separated into modules, and we want to use remote state with each of them, we would need to copy/paste the remote state backend configuration into the Terraform code for each of the modules, with only the key or prefix being different.

Terragrunt lets us keep the remote state configuration DRY by defining the remote state configuration once in the root level terragrunt.hcl file, and intentionally leaving each module’s backend configuration empty (this is known as partial configuration[[4]](#footnote-4) in Terraform).

#### 3.2.3 Executing Terraform on multiple modules at once

As our environments become more complex and we have our infrastructure resources separated into modules, it may be desirable for us (especially when first deploying an environment) to run Terraform against all of the modules at once, without having to run terraform apply for each module and wait for it complete before moving on to the next.

Terragrunt makes this process easy by providing a terragrunt run-all command, which we execute at the root level of the configuration tree. It supports the sub commands you would expect to run with Terraform: plan, apply, destroy, etc.

## 4.0 Continuous Integration / Continuous Deployment (CI/CD)

### 4.1 Code Repository

All infrastructure code will reside in a code repository. GitHub will be used to host the repositories, and we envision needing at least two repositories: one (or more) for Terraform modules, and another for environment live configuration using Terragrunt.

We will need to make a decision on how granular to get with repositories for Terraform modules, with the decision being based on the following factors:

1. The number and quality of public Terraform modules for Azure infrastructure
2. The need to pin module versions for known good/tested implementations
3. If using one module repository, how to source the modules in Terragrunt configuration

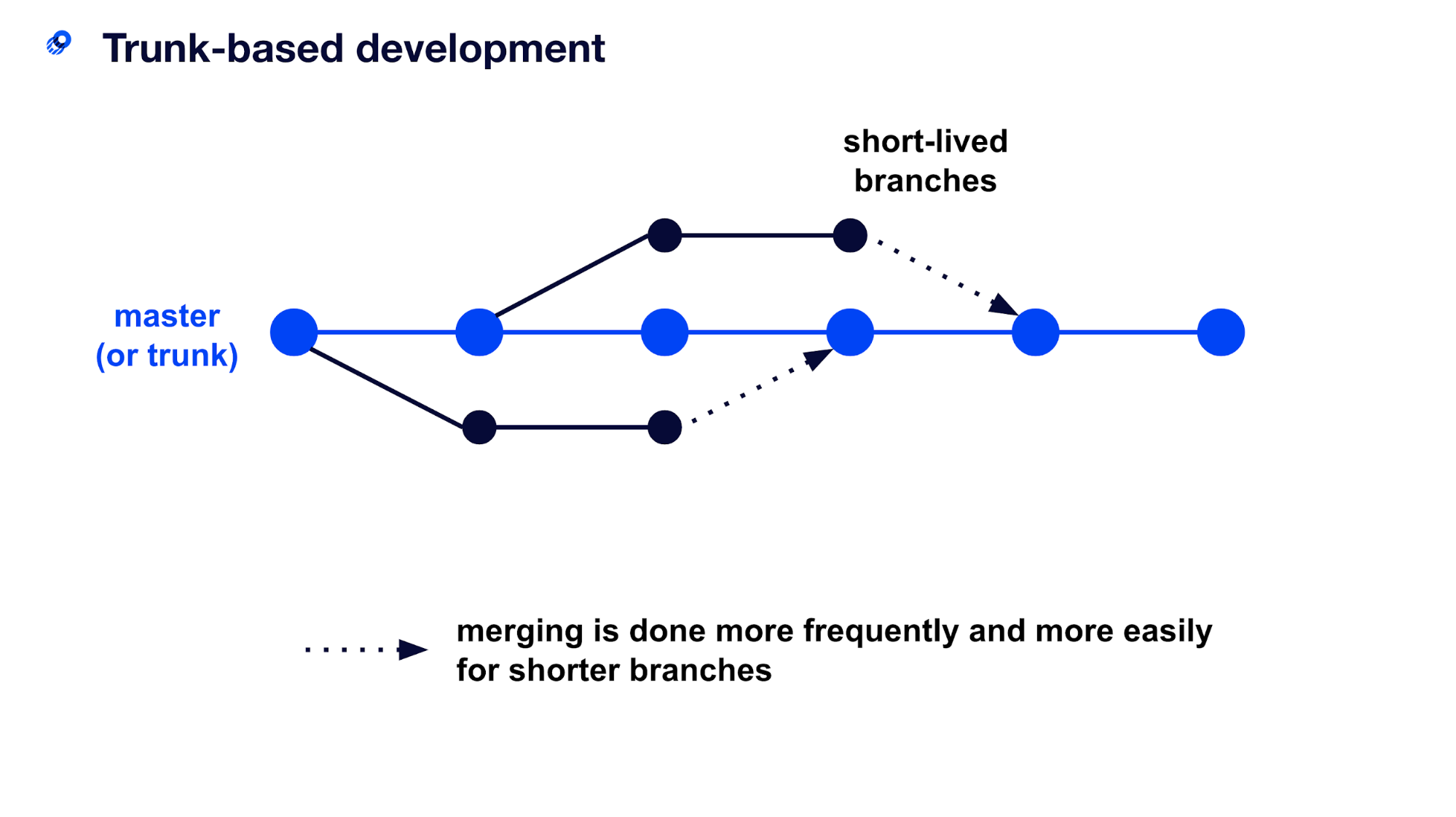
Unlike other cloud platforms that have provider-maintained modules and/or community modules which have had more time to mature, Terraform modules for Azure are still relatively new and we will likely need to develop and maintain our own modules. We will need to go through an exercise of evaluating the various infrastructure components that need to be built, and determining if there are public modules already available which can be used to manage them.

Similarly, the Azure cloud API is in rapid development and the Terraform azurerm provider and modules see many breaking changes. As a result, for any modules we develop and maintain, we will likely need to be able to pin versions for known good/tested implementations while allowing us to develop new functionality.

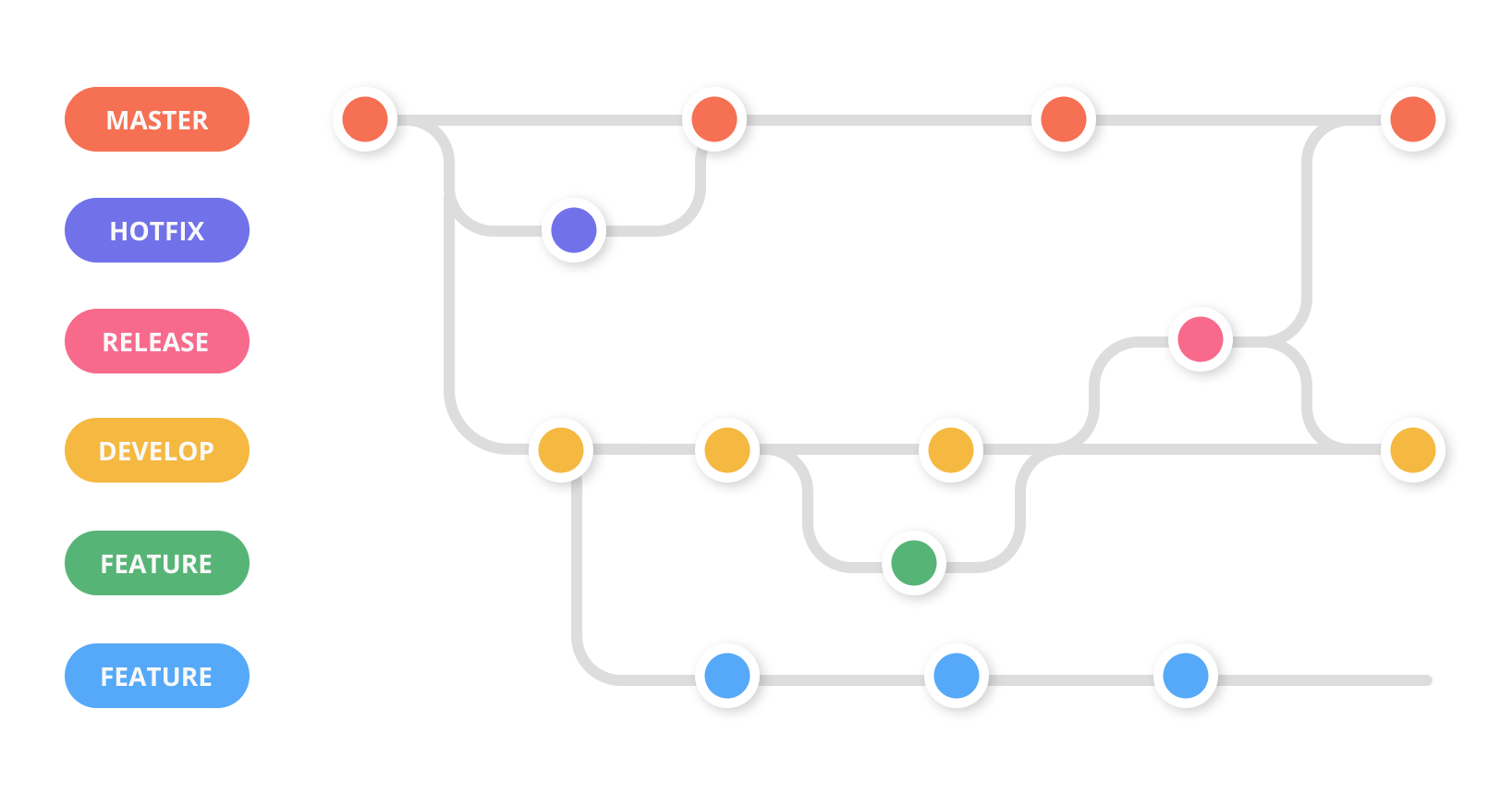
### 4.2 Development Branching Model (Trunk vs Gitflow)

There are two commonly used branching models for software development, ‘trunk’ and ‘gitflow’.

In trunk-based development there is typically a single main or ‘master’ branch, and development branches are short-lived before being merged back into the main branch.



With Gitflow, however, development branches for features and/or bugs are longer-lived, and changes are cherry picked to be merged into a versioned release.



For the repository containing the live configuration Terragrunt code, we recommend a trunk-based development model with the main branch set up as a protected branch, meaning developers cannot push commits directly to that branch. Instead, the process would be to create a development branch and push commits there, which are then peer reviewed using a pull request model before merging into the main branch.

For repositories containing Terraform module code, we recommend a Gitflow development model where feature/release branches are longer-lived. Changes will be merged into a versioned release of the module before they can be referenced in Terragrunt code.

### 4.3 CI/CD Pipelines

1. <https://www.terraform.io/docs/internals/graph.html> [↑](#footnote-ref-1)
2. <https://www.terraform.io/docs/language/state/index.html> [↑](#footnote-ref-2)
3. <https://www.terraform.io/docs/language/modules/index.html> [↑](#footnote-ref-3)
4. <https://www.terraform.io/docs/language/settings/backends/configuration.html#partial-configuration> [↑](#footnote-ref-4)