Mike Ensor

Follow 111 Followers About

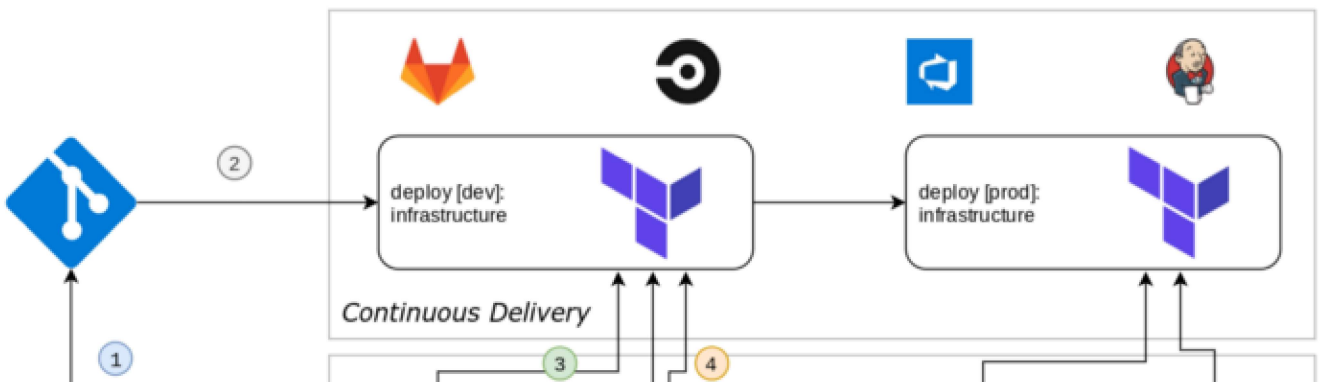
Terraform Developer's Tips & Tricks

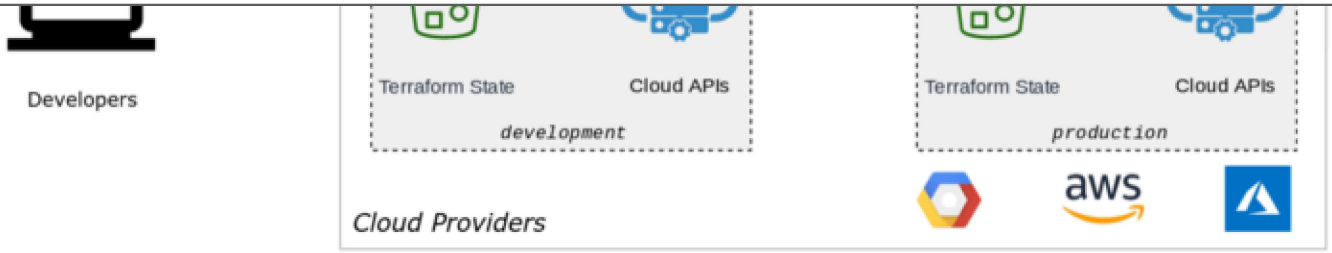
Mike Ensor Sep 18, 2019 · 7 min read ★

Terraform has become an industry standard tool to express infrastructure-as-code in a declarative format. Terraform can be complicated and often, advice changes based on your company and teams' unique experience level and adoption of Infrastructure-as-Code. A common best-practice discussed in most of the blogs states that Terraform should be run using a CI (continuous integration) tool. There are multiple challenges when developing while using a CI, this blog will cover **remote state**, **local vs CI builds** and **managing modules locally** based on experience learned over the last few years.

Ideal Development Process

In the ideal deployment process, developers make changes ① and submit pull requests ②. Upon acceptance, CICD pipelines use Terraform to validate the known Terraform state ③, formulate a plan against real infrastructure via APIs ④, execute the plan and update the Terraform state ⑤.





Ideal Terraform development workflow

The benefits of running Terraform through a CI are numerous, such as ability to code review before applying plans, avoid destroying critical systems, increase governance, provide self-service to teams while protecting layers of your infrastructure and many many more. That being said, developing with a CI comes with costs, primarily with inefficient use of developer time with misuse/lack-of-use with **remote state**, **long feedback cycle** involving many individuals, multiple repositories when using **modularity** and **parity between local-box and CI environment**. Terraform recommends the use of a CI only after the initial development of the infrastructure has been completed. The following will address these four issues.

Remote State

Using a remote state is absolutely critical to all terraform-based solutions. Remote state is a method in which Terraform stores what TF knows as the infrastructure state, along with metadata like "outputs", into a location outside of a single machine. In the above diagram, the remote state is stored in a "bucket" located within a cloud-provider's infrastructure. Storing the state outside of a single machine removes the coupling of a single machine to the solution as well as providing an accessible location for multiple systems and allows for backups. As with all distributed synchronous systems, there is a need for concurrency control, which is provided by Terraform via "lock files".

Using a remote state is rather simple, a developer configures the "backend" associated with the provider. This is very well documented within each of the Terraform provider's documentation. The documentation leaves many methods in which to initialize a provider, but does not specify a "best practice". When using two or more systems that both need to use the exact same remote state, a process needs to be developed to ensure the multiples of systems are working together.

```
    # resource_group_name  = "" #provided by -backend-var
    storage_account_name   = "<hard-coded-value>"
    container_name         = "__example-azure-devops-interpolation__"
    key                    = "${env.MY_KEY_ENV_VALUE}"
    # access_key           = "" #Provided by TF_VARS_access_key
  }
}
```

In this example, Azure is being used as a "remote state" and there are 5 **possible ways** of providing variables to the "backend", however, using "variables" or "tfvars" is not one of them due to Terraform having not been initialized, thus being a "chicken vs egg" scenario (see sample output below).

Variables need to be either **hard-coded**, use **interpolation** (but more complex but offers run-time variability), static **command-line arguments** during initialization or passed in via **Environment Variables** in combination with command-line variables.

```
± |feature/updating-ubuntu {1} U:3 ✗| → ~/bin/init-terraform.sh

Initializing modules...

Initializing the backend...

Error: Variables not allowed

  on provider.tf line 17, in terraform:
  17:     key                  = "${var.KEY}"

Variables may not be used here.
```

Example of using variables in Terraform Initialization

Four ways to Initialize Remote States

(1&2) Hard-coding and interpolation are very similar. Both require hard-coded values to be added to the source files which often violates security, compliance and company policies. Specifically, hard-coded values reduces audit & accessibility capabilities allowing actions to appear to be performed by a single user (shared-credentials).
(3) Using static strings, terraform has command-line parameters that can set variables for the "backend". Using parameters to specify each of the remote state backend variables is preferable to hard-coding or using interpolation as local and CI-based building have parity. However, the hard-coded strings often require a shared secret on each machine in order for the backend to reference the state bucket. `terraform init --backend-config="key="static-string-value""`

the repository, however, this process is an easy way to become complacent and commit sensitive data into a secondary repository. A recommended scenario would be to auto-generate files at run-time. The caveat being that a file still exists potentially exposing sensitive information.

(5) A combination of using command-line variables and environment variables is the final method of injecting dynamic parameters into Terraform state, while increasing relevant parity between CI and other systems, without sharing or reusing identity/access keys. The act of setting environment variables can be decoupled from the use of the variables as shown in the 6th line of the below example. This script should be checked into the repository allowing both the CI and the developer(s) to operate against the same bucket, while allowing each entity to designate their own access keys for authentication.

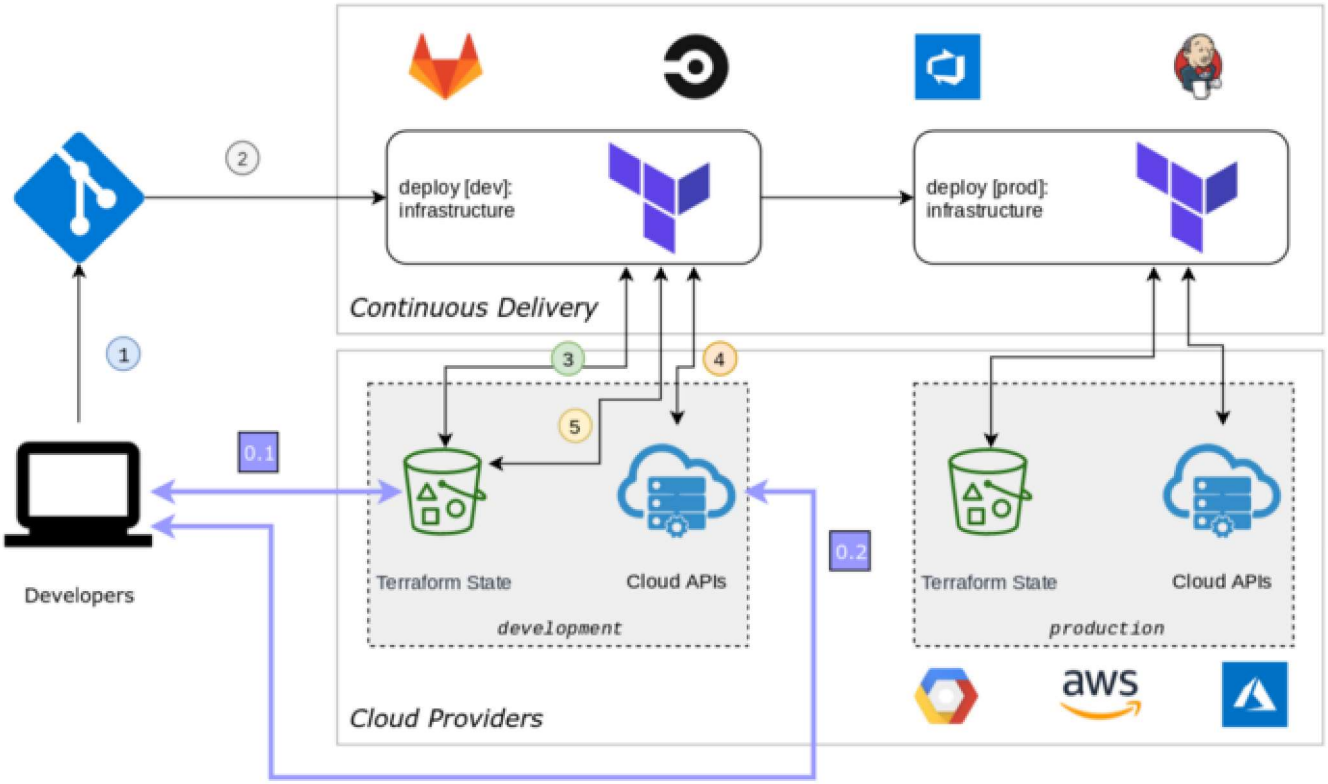
⇒ Pro Tip: modern CI tools allow for the use of secret or priviledged environment variables that become masked in all logs. Additional security measures can be taken if using a Secrets Management tool like Vault.

Example shared Terraform Initialization Script

```
#!/bin/bash

required_vars=( "SUBSCRIPTION" "RESOURCE_GROUP_NAME"
"STORAGE_ACCT_NAME" "CONTAINER_NAME" "ACCOUNT_KEY" "KEY" )

source ~/.terraform-state-variables.sh # EX EXTERNAL VAR SETTING

for each in "${required_vars[@]}"
do
    var="\$$each"
    # echo "${var}"
    if ! [[ -v $each ]]; then
        echo "ERROR: ${each} ENVVAR not defined & required"
        exit 1
    fi
done

terraform init -reconfigure \
    -backend-config="subscription_id="${SUBSCRIPTION}"" \
    -backend-config="resource_group_name="${RESOURCE_GROUP_NAME}"" \
    -backend-config="storage_account_name="${STORAGE_ACCT_NAME}"" \
    -backend-config="container_name="${CONTAINER_NAME}"" \
    -backend-config="access_key="${ACCOUNT_KEY}"" \
    -backend-config="key="${KEY}""
```

recommend using `-reconfigure` as a part of `terraform init`.

Developing Locally vs Building with CI



Recommended development workflow

When developing the infrastructure for the first time, there are going to be bugs, trials, tests, and many things won't work the first time. In order to facilitate faster development times, it is wise to skip the CI on a lower-level environment like `development`. This process is in-line with other forms of software delivery where-as other languages compile and build locally, Terraform would need to replicate this concept. NOTE, while building infrastructure from a development machine is simple, a best practice is to have the Continuous Integration server apply or promote the changes to each environment after the development has been completed. It is unwise to develop production Terraform and run from one or more development machines as this does not follow best practices for Continuous Delivery.

Typical Development Workflow

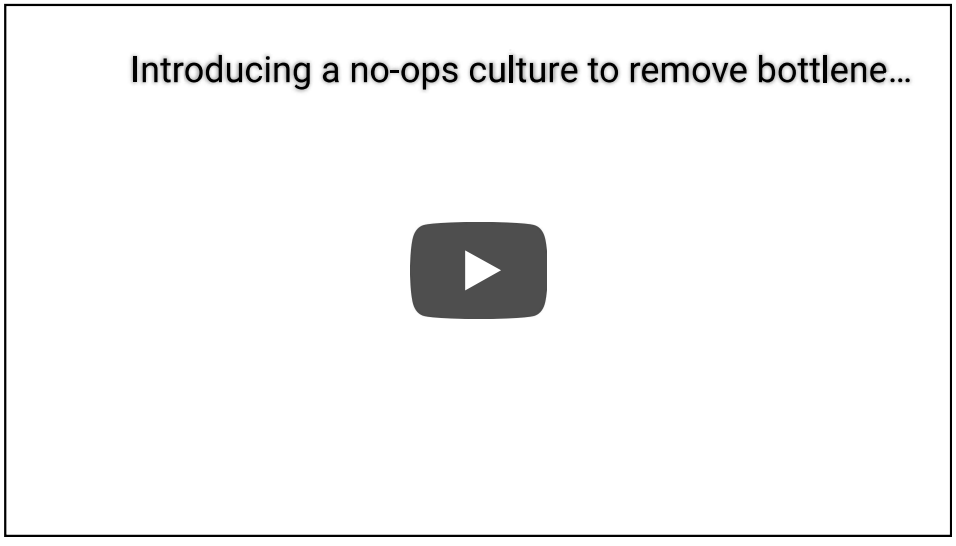
The diagram above shows the typical workflow of a developer building Infrastructure-as-Code. Using the last example in the "Remote State" section above (5), developers can

unit tests and validate functionality over a series of iterations. Once verified and tested, the developer can optionally destroy destroy the infrastructure and issue a pull-request for others to view the changes. Developers do not have to destroy the infrastructure as the new state and cloud infrastructure will be consistent, however, running changes through the CI could identify issues with missing variables or issues coming from characteristics local to the development machine.

Local Development & Modules

Developing modules is Terraform's method for DRY programming. Modules provides the ability to encapsulate a logical group of resources into a single action. Variables are exposed and act like interface variables, while `outputs.tf` files provide meaningful outputs. Creation of modules should represent a group of resources that can be reused and should follow the Goldilocks principle (not too big, not too small, just right).

There are two primary methodologies that Terraform repositories are typically implemented with. The first methodology is to put all infrastructure code into one repository. This methodology tends to follow organizations where there is lower overall trust of the development teams and IT maintains control. The second methodology is the opposite, enabling self-service through the use of many terraform repositories (or terraform within application projects). These organizations typically build up layers of terraform repositories and only expose functionality typical development teams would need resources for, like databases, storage buckets and queues.



This blog will not go into the details of either, but the methodologies are mentioned due to the availability of `module` in Terraform code.

Local Development of Remote Modules

Developing modules can be difficult if the end-result is to publish modules to a git repository for a wider adoption within your private enterprise, or even to public. Typical development typically ends up as developer changes and pushes the module code, then follows up by `terraform init` after deleting the `.terraform/` folder or using `-reconfigure` option. The round trips to change, re-publish, pull and verify impact within the local development is not only painful, it's time consuming. There is a better way.
⇒ Pro Tip: When Terraform initializes the folder, remote modules are pulled down as a cached version. The cached version is used by Terraform during `plan` and `apply`, so modifying this cached version allows development to test the changes locally, before applying changes on the remote module.

Conclusion

Terraform is a key tool in modern solution development. Usage has continued to grow year-over-year and will continue to incorporate feedback from developers world-wide. This blog has identified 3 areas/concepts to make development just a bit less difficult. At the time of this blog, Terraform 0.12 had just been released and the shared tips does not include use of Terragrunt or other Terraform wrapper tools. Please leave some comments so we can all help each other become more efficient, or remove some of the pains associated with cutting-edge technology.

DevOps Terraform Noops Infrastructure Software Development