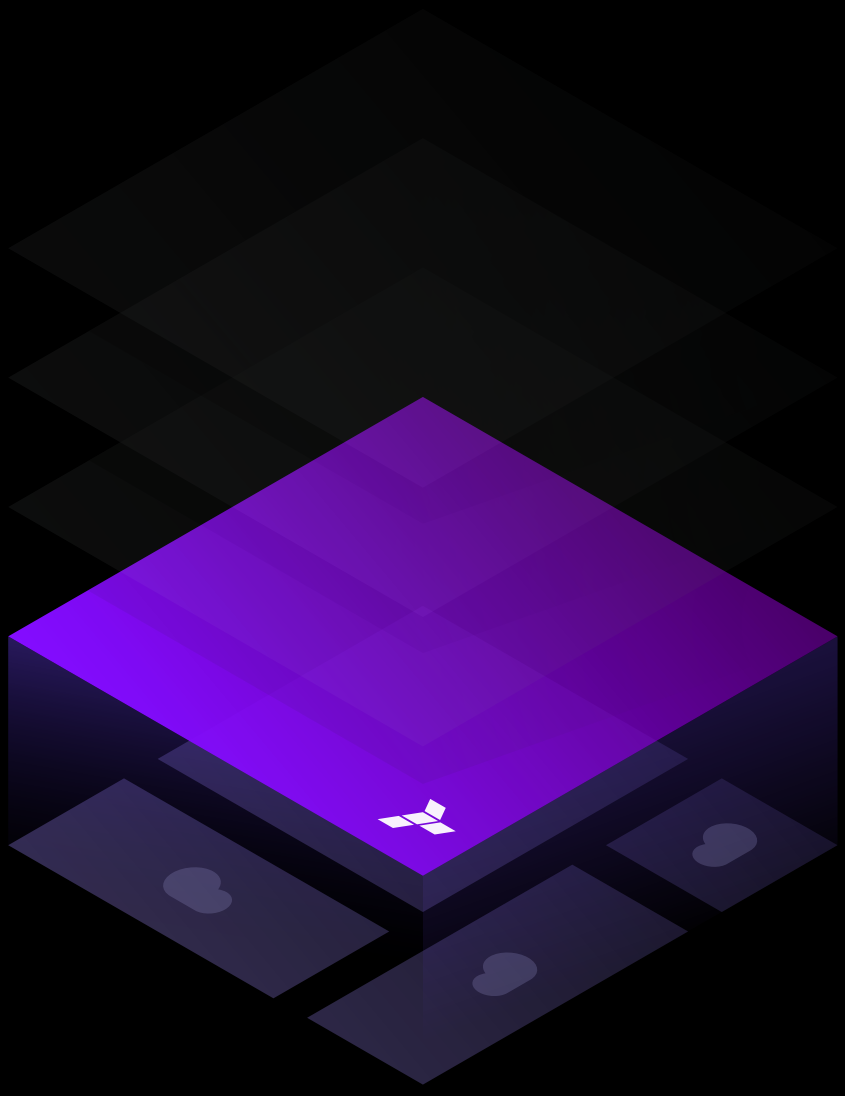
HashiCorp Terraform | GitHub

A Practitioner's Guide to Using HashiCorp Terraform Cloud with GitHub



HashiCorp

Executive Summary

HashiCorp Terraform and GitHub's integrations form the foundation for a DevOps workflow that moves at the speed your organization desires. This speed is often a response to the velocity GitHub provides the development teams they serve. Developers are now able to automatically integrate changes to code from anywhere, and then deploy those changes in any environment on demand.

This presents a challenge for operations teams to have the required infrastructure ready at the time that it is needed. While it is possible to meet this challenge by leaving instances active at all times, the better and more cost-effective solution is to make the infrastructure a part of the versioning process and add or remove resources on-demand. By tying application development to underlying infrastructure, GitHub and Terraform solve the provisioning challenge to keep developer velocity up and cloud costs down.

This guide illustrates the various approaches to configure a continuous integration and continuous delivery (CI/CD) workflow using GitHub and Terraform Cloud to address the challenges of dynamic DevOps environments. It covers repository management, setting up a CI/CD pipeline, available areas of integration, and some of the security considerations when using Terraform.

For more information on how HashiCorp tools and the GitHub platform work together to enable organizations to adopt a strong CI/CD workflow, see: Increasing Developer Velocity in the Cloud Operating Model.

Understanding Terraform and GitHub's Role in a GitOps Workflow

Terraform and GitHub are central components of a GitOps workflow. Terraform enables teams to codify infrastructure as configurations — a concept better known as "infrastructure as code". GitHub provides a central source of truth and version control for this infrastructure code in the same way that it would for application code. This combination enables operations teams to work with the same speed as the development teams they serve.

Deploy Infrastructure as Code with Terraform

Terraform provides a single workflow for building, changing, and versioning infrastructure safely and efficiently. It connects to hundreds of cloud services as well as private infrastructure, and enables automated provisioning. Terraform uses an easy-to-understand configuration language called HCL that allows practitioners to provision and manage infrastructure as code, with cleaner syntax than alternatives like YAML. Using just the Terraform CLI on their local machine, an individual practitioner could manage something as simple as a single application or as complicated as your entire cloud infrastructure.

HashiCorp offers two products that help Terraform users gain the full benefits of a GitOps workflow: Terraform Cloud and Terraform Enterprise. Terraform Cloud is available as a fully managed service, and Terraform Enterprise can be installed privately in an organization's datacenter. These Terraform editions execute runs, at scale, in a consistent and reliable environment. Plus, they include key team collaboration and governance features.

Terraform Cloud is optimized for DevOps collaboration while also giving your organization advanced and granular compliance features. We'll be going over how their features enable best practices throughout this white paper.

If you happen to be unfamiliar with Terraform Cloud, there is one particular concept which needs to be defined ahead of time and that is workspaces. Workspaces, when it comes to the Terraform CLI, are used in a way to create multiple state files based on similar Terraform configuration files to reduce directory sprawl. As an example, these state files can differentiate based on which variables are called as part of the Terraform operations. On the other hand, when dealing with Terraform Cloud, workspaces are how Terraform configurations are organized and managed. These types of workspaces contain all the requirements that Terraform will need to manage the declared workloads.

GitHub Provides a Source of Truth and Collaboration for Infrastructure

GitHub is a complete DevOps and collaboration platform that is best known for its version control capabilities. Version control systems (VCS) are generally used to store a collection of software files, making it possible to document, track, undo, and combine changes all made in parallel by different users. Through discussions that take place in pull requests and issues, GitHub also acts as a collaboration platform for millions of developers. In using GitHub for version control and for collaboration, operators can more easily coordinate with application developers throughout the software lifecycle.

Terraform users should store their configuration files in a VCS repository. Storing infrastructure as code in a VCS allows them to version control, collaborate and continuously improve infrastructure as code as a delivery pipeline.

Best Practices with Terraform Configurations and GitHub

Terraform Cloud integrates tightly with GitHub. As changes are committed to a repository that affects the attached Terraform organization, Terraform can automatically initiate a run and make the necessary modifications to accommodate the change. Operators can also use this integration to version their various infrastructure environments. If more VMs are required for a specific event or test, these changes can be made to the Terraform configurations stored in GitHub. After the event has completed or if an issue arises from the deployment, operators can roll back the infrastructure to a previous, stable version.

Repository Management

Terraform configurations can be defined in a wide variety of ways. The most straightforward Terraform configuration is a single root module containing only a single .tf file. A configuration can grow gradually, as more resources are added, either by creating new configuration files within the root module or by organizing sets of resources into child modules. Therefore, structuring these repositories properly is important because it determines which files Terraform has access to when the specified Terraform operations are executed.

Structuring Repositories for Multiple Environments

When each repository represents a manageable chunk of Terraform code, it is often still useful to attach a single repository to multiple workspaces in order to handle multiple environments or other cases where similar infrastructure is used in a different context. There are three primary ways to structure the Terraform code in your repository to manage multiple environments (such as: development, staging, and production).

Depending on your organization's use of version control, one method for multi-environment management may be better than another.

Multiple Workspaces per Repository

Using a single repository attached to multiple workspaces is the simplest approach. This enables the creation of a pipeline to promote changes through environments without additional overhead in version control. When using this model, one repository is connected to multiple workspaces such as prod, stage, dev. While the repository connection is the same in each case, each workspace can have a unique set of variables to configure the differences per environment.

To make an infrastructure change, a user opens a pull request on the specified repository. After the pull request has been merged, they can then apply it in Terraform Cloud one workspace at a time. First by starting with the dev workspace, progressing to stage, and ending the process with the prod workspace.

This model will not work for a repository if there are significant environmental differences between the Terraform configurations. For example, if the prod workspace has 10 more unique resources than the stage workspace. They likely cannot share the same Terraform configuration and thus cannot share the same repository. If this is the case for a given repository, one of the other approaches may be better.

Single Workspace per Repository Branch

For those organizations with a preference for long-running branches, there is an option to create a branch for each environment. When using this model, one repository could have three long-running branches such as prod, stage, and dev. Using the branch strategy reduces the number of files needed in the repository.

In the example repository structure below, there is only one main.tf configuration and one variables.tf file. When connecting the repository to a workspace in Terraform Cloud, you can set different variables for each workspace. This means there is one set of variables for prod, one set for stage, and one set for dev.

```
├── README.md
├── variables.tf
├── main.tf
├── outputs.tf
├── modules
│   ├── compute
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── networking
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
```

Each workspace listens to a specific branch for changes, as configured by the GitHub branch setting. This means that plans will not occur in a given workspace until a pull request is opened or a push event occurs on the designated branch. Therefore the prod workspace would be configured to listen to the prod branch, stage to stage, and dev to dev. To promote a change to stage, open a pull request against the stage branch. To promote to prod, open a pull request from stage against prod.

The upside of this approach is that it requires fewer files and runs fewer plans, but the potential downside is that the branches can drift out of sync. It is imperative in this model to enforce consistent branch merges when promoting changes.

Single Workspace per Repository Directory

For organizations that have significant differences between environments, there is an option to create a separate directory for each environment. This option could also apply when short-lived branches that are frequently merged into the master branch are preferred.

In the example repository structure below, the prod, stage, and dev environments have separate main.tf configurations and variables.tf files. These environments can still refer to the same modules (like compute and networking).

```
├── environments
│   ├── prod
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── stage
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── dev
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
├── modules
│   ├── compute
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
│   ├── networking
│   │   ├── README.md
│   │   ├── variables.tf
│   │   ├── main.tf
│   │   ├── outputs.tf
```

When using this model, each workspace is configured with a different Terraform working directory. This setting tells Terraform Cloud which directory to execute Terraform in. The prod workspace is configured with prod as its working directory. The stage workspace is configured with stage as its working directory, and likewise for dev. Unlike in the previous section, every workspace listens for changes to the master branch in the specified directory.

The potential downside to this approach is how changes have to be manually promoted between stages. This means the directory contents can potentially drift out of sync.

GitHub Integrations with Terraform Cloud

Terraform Cloud integrates directly with GitHub. This integration provides a streamlined and seamless workflow for practitioners to store and manage their code directly in GitHub while bringing a new level of automated functionality to their process.

Configuring GitHub as a Version Control System

Connections between Terraform Cloud and GitHub can be established in several different ways. For those accessing repositories through GitHub.com, Terraform Cloud offers both the configuration-free GitHub App or an OAuth-based connection.

Through the GitHub App

The configuration-free GitHub App can be used in a wide variety of use cases and is considered to be the most secure and flexible. It is also the easiest way to connect GitHub to Terraform Cloud. Integrating with GitHub in this manner will require each Terraform Cloud user to authenticate and authorize their GitHub account and resources for usage in the Terraform Cloud organization.

It should be noted how this may introduce some complexities for collaborative environments where users may not have access to the same GitHub repositories. For example, a Terraform Cloud user will not be able to run a plan operation if their GitHub connection does not have access to the specified repository. It is also important to keep in mind how there may be some limitations, or inaccessibility, when using this method with Terraform Cloud-based private module registries (discussed below) and when attempting to create additional workspaces through the Terraform Cloud API.

Through the OAuth Connection

The GitHub OAuth connection establishes a link at the Terraform Cloud organization level. The GitHub OAuth connection also creates the link as one particular GitHub user. This helps provide a consistent level of access and permissions for everyone in a given Terraform Cloud organization. During the process of establishing the connection, a user will be asked to create a new GitHub-based OAuth application and provide the Client ID and Client Secret to Terraform Cloud. Optionally, an SSH keypair can also be configured where required.

On-premises instances of GitHub Enterprise will also use OAuth connections. However, while the process is very similar to the prior option, there will be an additional setup step to provide the local HTTP and API URLs of their Terraform instance.

Pull Request Integrations

Once you've established the integration between your Terraform Cloud workspace and your GitHub repository, Terraform Cloud will automatically perform what's known as a 'speculative plan' any time a pull request is created against the connected repository. These speculative plans act as integrated checks to your GitHub repository, and the results of these plan-only runs will be visible on the associated pull request. From there, the results can be integrated into your existing GitOps workflows or simply used to let you see exactly what happens if you merge the pull request. Speculative plans also take into consideration the outcome from the applied Sentinel rules and policy sets and presents them back through to the pull request system.

Committed Code Integrations

Terraform Cloud can also be notified any time the repository sees a successful code commit. By default, each commit to the configured repository will result in a plan operation being performed. A Terraform Cloud user with access to the workspace can then apply or cancel the run operation from being completed.

This action is due in part to the GitHub integration, but also due to the Apply Method configuration. This configuration is applied at the workspace level and features two settings. The first, which is the default, is Manual Apply. The second setting, Auto Apply, will track each commit to the repository, run the plan operation, and, assuming all Sentinel rules and policy sets are successful, will apply the new configuration automatically without any user intervention required.

GitHub Actions

GitHub Actions gives users the ability to configure custom workflows based on nearly any event in the GitHub ecosystem, such as pull requests, comments within issues, and merges to repositories. This feature can be used for Terraform modules managed in GitHub without having to rely on any external tooling.

GitHub Actions relies on a YAML workflow file to specify the steps to execute. A typical workflow for a Terraform module includes **terraform init** and **terraform validate** commands. The **init** command initializes the module and downloads any needed providers. The **validate** command helps validate the configuration files in the module and is useful for general verification.

In addition to validating Terraform configurations, we can extend this workflow to incorporate automated testing of any part of our configuration, including modules. This is especially important when multiple developers are collaborating on a particular module, and helps continuously verify that it executes as expected.

And finally, operators can use GitHub Actions to easily coordinate with application development teams. For more, watch the webinar Unlocking the Cloud Operating Model with GitHub Actions.

Security Considerations

Terraform configurations could eventually be the single source of truth for the state of your entire organization's cloud infrastructure. It is important to carefully consider security implications, particularly in a dynamic organization where infrastructure and possible staff changes could be a regular occurence.

Like GitHub, Terraform Cloud features role-based access controls and advanced permissions. It is important to consider all of the access levels available with Terraform.

Gitignore Considerations

One of the easiest ways to secure your Terraform configuration is to ensure files containing potentially private information, such as API tokens, account information, cloud resources and their identifiers, and so forth, are not inadvertently stored in your VCS repository. These sensitive files can be excluded from version control through the usage of the gitignore file.

The gitignore file can be used to tell Git what files in the repository should not be tracked. Some examples of items that should not be added to version control would be state files, variables files with the .tfvars extension, the entire .terraform directory, and crash log files. An example gitignore file is available on GitHub and should be added to every Terraform repository: Terraform.gitignore.

Terraform State File Storage

A Terraform state file maps real world resources with the resource definitions within an organization's declarative terraform configuration files. It's important to note that state files often contain sensitive information which should not be shared publicly or stored in version control. That said, there are numerous locations where state files may be safely stored, but each option has different implications when it comes to access and security.

One of the easiest methods to secure state file storage is by leveraging remote state storage in Terraform Cloud. Remote state storage allows an administrator to designate specific outputs to only the teams that need to see them. Another benefit of remote state storage within Terraform Cloud is state locking. This prevents any potential overlap where two teams may be accessing and changing the same infrastructure at the same time.

Alternatively, remote backends can be used to collaborate and maintain sensitive information. These backends can point to a wide variety of cloud-based storage services, like Azure Blob Storage. When using remote backends, the accessibility and security over the state files are transferred from Terraform to the storage services themselves.

The default method of state file storage when using Terraform CLI is known as a local backend. This configuration stores the state in the same directory as the Terraform configuration itself. There are security implications when configurations, especially with state files, are being stored in a VCS repository.

Private Module Management

Private module management is an exclusive feature within Terraform Cloud. The private module registry gives organizations a way to codify modular templates that can be used across their environment. These templates, which are known as modules, are customized and validated based on the organization's particular requirements. These modules are published directly from their GitHub repositories into the service catalog. Once in the service catalog, the modules are associated with searchable metadata to make them easy to find and use throughout the environment's infrastructure. This workflow enables every team within an organization to safely and efficiently provision infrastructure.

Permissions Management and Access Control

There have been numerous mentions of cases where users without access to Terraform Cloud and Enterprise could perform updates to the infrastructure through updates to the GitHub repository. This can also be applied to users without access to the GitHub repository, yet do have access to Terraform Cloud. Therefore it is important to understand the levels of privilege throughout the environment.

Terraform Cloud has three levels of access control: Users, teams, and organizations. Users are individual accounts which are part of an organization. Teams are groups of users which have particular access configured within an organization. Organizations are where users can collaborate on a shared set of workspaces. Teams allow for two options of access within the organization. These two options are read — which is set by default — and manage. Manage allows users to create, edit, read, and remove Sentinel policies, workspaces, and VCS settings.

Each workspace has their own set of permissions that can be utilized. These permissions are configured against the available teams as either fixed permission sets or through more granular rights. The permission sets are listed as Read, Plan, Write, and Admin. Read allows members of the specified team to read workspace information such as variables, runs, and state. The Plan set builds on the read permissions with the ability to create run operations. The Write set continues from the Plan set and allows the approval of runs, configuration of the workspace's lock status, and the ability to update variables and state versions. Lastly, the Admin set allows for full workspace capabilities including the configuration of team access, deletion of the state, and execution mode.

On the other hand, when it comes to permissions on the GitHub repository, there is a certain set of permissions to be aware of. Any permissions on repositories which result in code being committed could potentially have unintended consequences. These include users having the ability to push code directly to a repository or merge pull requests. These permissions are particularly important for those Terraform Cloud workspaces using auto apply, though could also be important when manual apply is in use.

Conclusion

There are many features within Terraform Cloud and GitHub that increase DevOps velocity while also providing the required checks and guardrails for security and compliance. To set up your own GitOps pipeline, start by signing up for Terraform Cloud for free. Find out more about version control system integrations, see the webinar VCS + Terraform Cloud. To walk through an example on organizing Terraform configurations, use the Learn Guide for Organize Configuration. For more information about automating workflows with GitHub Actions, see the blog Automate Infrastructure Provisioning Workflows with the GitHub Action for Terraform and the webinar Unlocking the Cloud Operating Model with GitHub Actions.


HashiCorp