Terraform has become the standard for managing infrastructure as code (IaC). GitHub Actions is a continuous integration and delivery (CI/CD) platform integrated into GitHub.

In this blog post, we will explore managing infrastructure with Terraform and GitHub actions. We will show how to combine them to orchestrate our infrastructure workflows and outline the benefits and caveats of this approach.

If you aren’t familiar with Terraform, look at the [content-rich Terraform section](https://spacelift.io/blog/terraform) of Spacelift’s blog.

We’ll cover the following:

1. [What is Terraform?](https://spacelift.io/blog/github-actions-terraform#what-is-terraform)
2. [What is GitHub Actions?](https://spacelift.io/blog/github-actions-terraform#what-is-github-actions)
3. [How to use Terraform in GitHub Actions – Example](https://spacelift.io/blog/github-actions-terraform#how-to-use-terraform-in-github-actions--example)
4. [Tips for troubleshooting GitHub Actions workflows](https://spacelift.io/blog/github-actions-terraform#tips-for-troubleshooting-github-actions-workflows)
5. [Terraform in GitHub Actions: Best Practices](https://spacelift.io/blog/github-actions-terraform#terraform-in-github-actions-best-practices)
6. [Managing Terraform with CI/CD **—**scaling and operational concerns](https://spacelift.io/blog/github-actions-terraform#managing-terraform-with-cicd--scaling-and-operational-concerns)
7. [How Spacelift helps with managing Terraform at scale](https://spacelift.io/blog/github-actions-terraform#how-spacelift-helps-with-managing-terraform-at-scale)

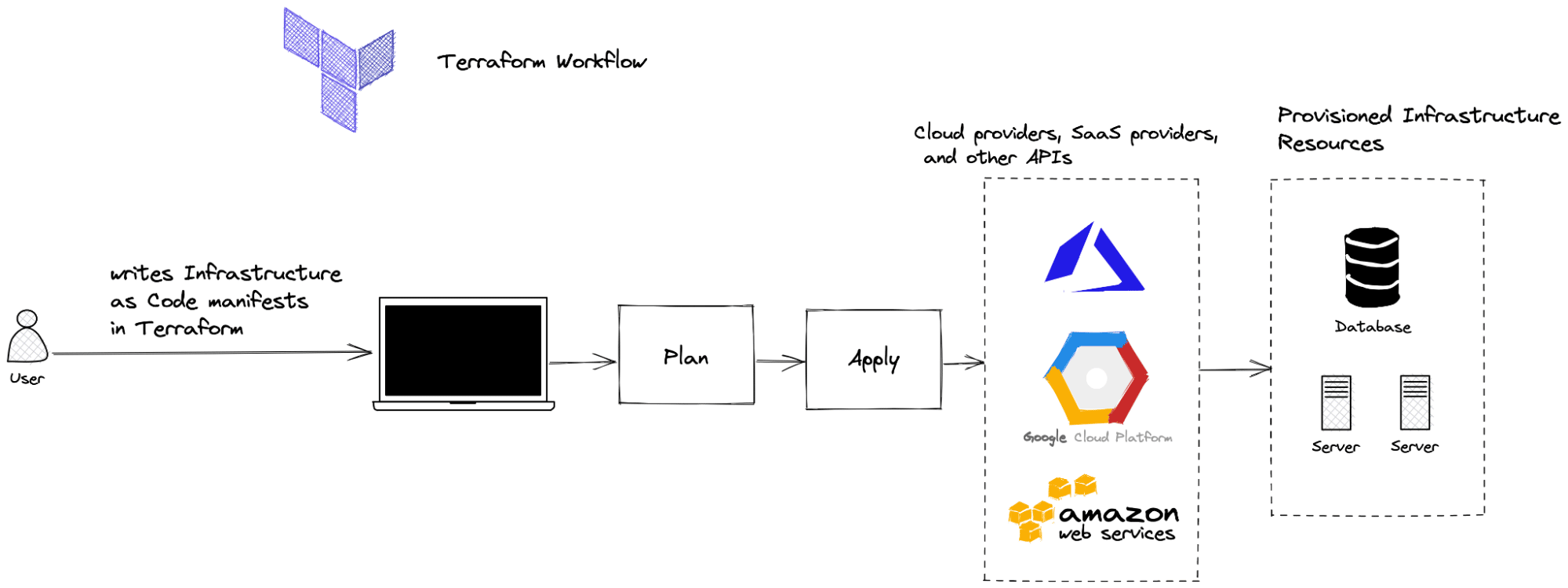
**What is Terraform?**

Terraform is an IaC software tool that allows us to safely and predictably [manage infrastructure at scale](https://spacelift.io/blog/5-ways-to-manage-terraform-at-scale). Widely adopted by organizations and IT professionals over the years, it is recognized as one of the most influential tools in the space. Its cloud-agnostic characteristics, IaC principles, modularity concepts, and automation capabilities make it a powerful tool that facilitates infrastructure management in any environment.

Terraform keeps and manages an internal state of its managed infrastructure and is used to create plans, track changes, and enable safe modifications in live environments. One of the reasons for Terraform’s success has been its intuitive and easy-to-understand workflow.

**Terraform workflow**

The core Terraform workflow consists of three concrete stages. First, we generate the IaC configuration files representing our environment’s desired state. Next, we check the output of the generated [plan](https://spacelift.io/blog/terraform-plan) based on our newly edited manifests. After carefully reviewing the changes, we [apply](https://spacelift.io/blog/terraform-apply) the plan to provision the infrastructure changes and resources.



**Note:** New versions of Terraform will be placed under the BUSL license, but everything created before version 1.5.x stays open-source. [OpenTofu](https://opentofu.org/" \t "_blank) is an open-source version of Terraform that will expand on Terraform’s existing concepts and offerings. It is a viable alternative to HashiCorp’s Terraform, being forked from Terraform version 1.5.6. OpenTofu retained all the features and functionalities that had made Terraform popular among developers while also introducing improvements and enhancements. OpenTofu is not going to have its own providers and modules, but it is going to use its own registry for them.

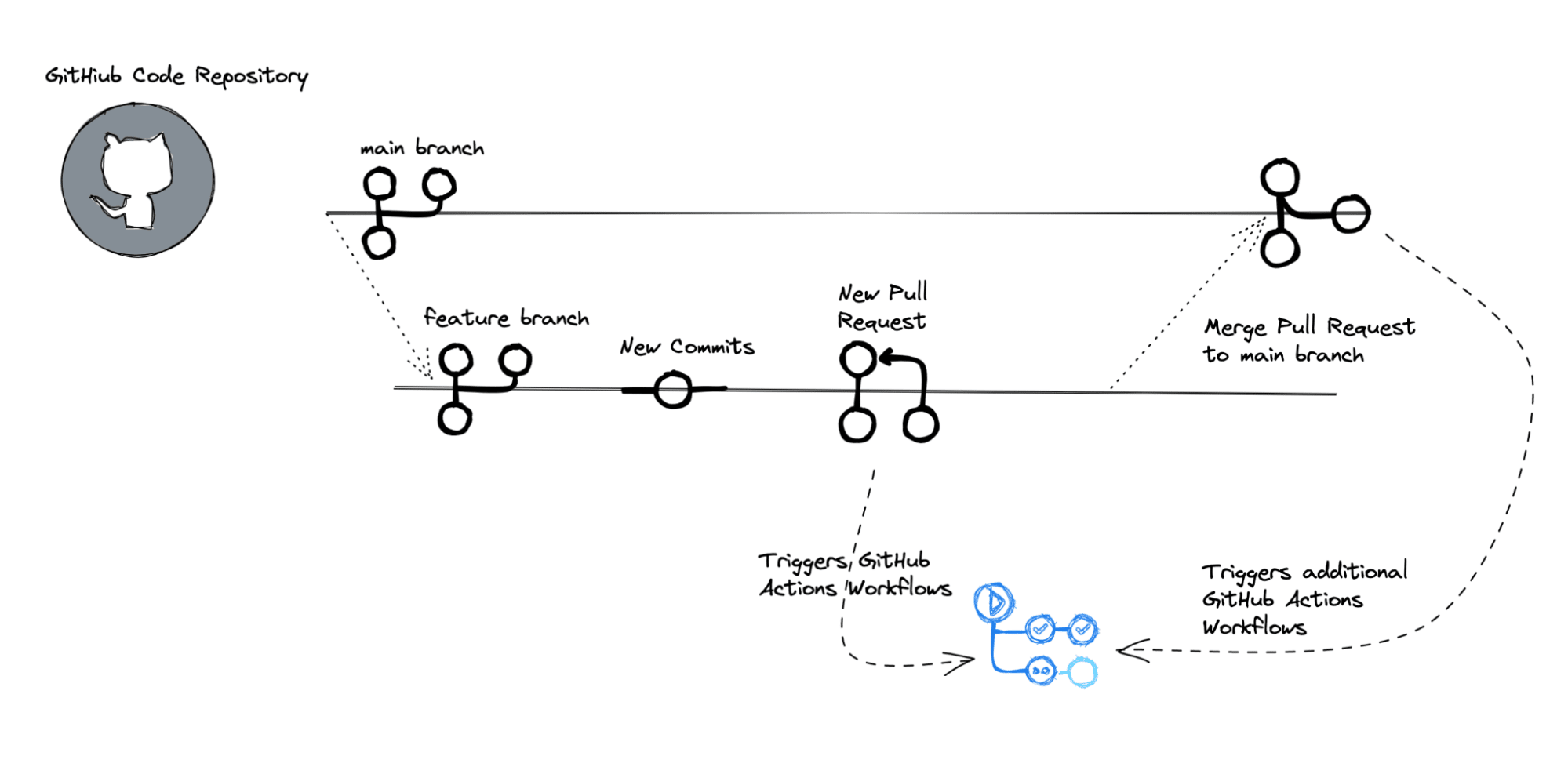
**What is GitHub Actions?**

[GitHub Actions](https://github.com/features/actions) is a modern CI/CD tool integrated natively on GitHub. Itenables the rapid automation of build, test, deployment, and other custom workflows on GitHub with no need for external tools.

It is design to provide an easy and seamless way to automate every software workflow right from GitHub while providing and abstracting all the necessary infrastructure pieces. The tool’s architecture depends heavily on events that trigger further actions combined to generate custom user-defined workflows.

Over the years, GitHub Actions has evolved into a mature tool in the CI/CD ecosystem and provides many customization options and a robust workflow engine if your team is already using GitHub. It offers options for customizing runners or bringing your own VMs, support for workflows across different environments, operating systems, versions, and programming languages.

A major benefit of using GitHub actions is the open-source community-powered workflows available on GitHub to get you up and running quickly.



Check out the [official GitHub Action docs](https://docs.github.com/en/actions) and the [Learn GitHub Actions](https://docs.github.com/en/actions/learn-github-actions) section for more information and examples.

**How to use Terraform in GitHub Actions - Example**

To create an automated infrastructure management pipeline with GitOps principles, we can combine GitHub, GitHub Actions, and Terraform. The first step is to store our Terraform code on GitHub. Then, we configure a dedicated GitHub Actions workflow based on our needs that handles infrastructure changes by updating the Terraform configuration files.

Let’s outline an example of such a setup.

**1. Prerequisites**

Structuring Terraform code repositories is a separate topic we won’t go into in this post. Also, for this simple Terraform example, I ignore many best practices, such as using variables, modules, separating environments, etc., to focus on the GitHub Actions workflow.

Check out these articles if you are looking for inspiration for a [production-grade Terraform setup](https://spacelift.io/blog/terraform-best-practices) or [configuring Terraform in automation](https://spacelift.io/blog/terraform-automation).

**2. Create a GitHub repository**

We will set up a GitHub repository with a simple Terraform file that deploys an EC2 instance on AWS. You can [find the demo contents here](https://github.com/spacelift-io-blog-posts/Blog-Technical-Content/tree/master/terraform-github-actions).

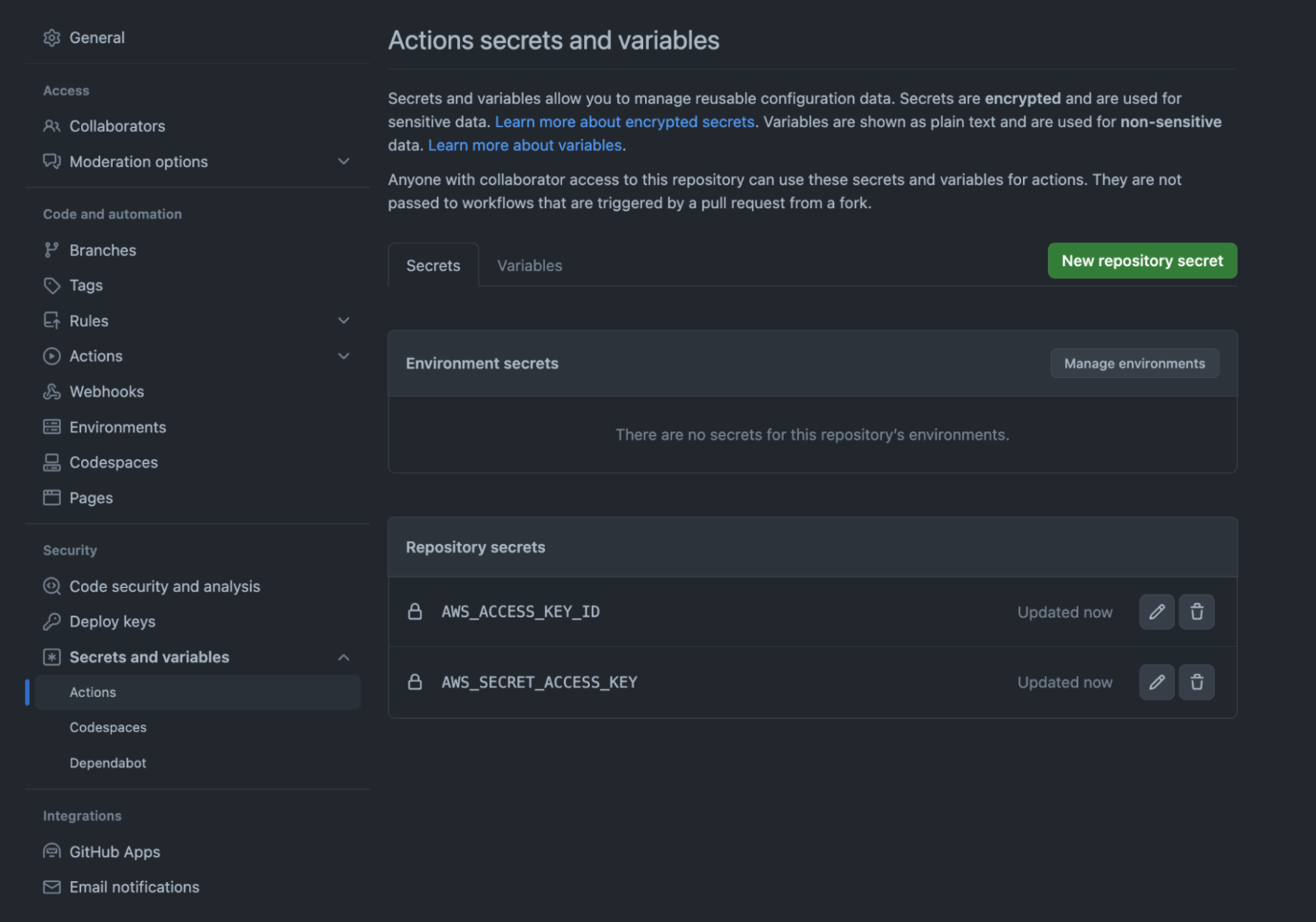
**AWS provider**

AWS credentials can be configured in many ways, but for this example, we will use the most basic configuration — static credentials.

For this to work, you will need to go to your Repository → Settings → Secrets and variables → Actions. Here, you have to select the New repository secret option and configure the following secrets:

* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY

The screenshot below shows the secrets after they are configured:



**Remote state**

In your Terraform code, you won’t need to specify anything for your provider, apart from the region, as it will know how to get these values from the environment.

provider “aws” {

region = “eu-west-1”

}

Configuring a remote backend that leverages s3 can be done in the following way:

terraform {

backend "s3" {

bucket = "my-terraform-state-bucket"

key = "path/to/my/key"

region = "eu-west-1"

}

You can also enable state locking if you are using a DynamoDB table, bearing in mind that the Key Schema Attribute name should be **LockID**.

terraform {

backend "s3" {

bucket = "my-terraform-state-bucket"

key = "path/to/my/key"

region = "eu-west-1"

dynamodb\_table = "my-terraform-lock-table"

encrypt = true

}

**Terraform configuration files**

Our Terraform file *main.tf* contains provider configuration, backend state configuration with an S3 bucket, and a simple ec2 instance. This is nice and simple for the purposes of our demo.

*main.tf*

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

}

}

backend "s3" {

region = "us-west-2"

key = "terraform.tfstate"

}

}

provider "aws" {

region = "us-west-2"

}

resource "aws\_instance" "test\_instance" {

ami = "ami-830c94e3"

instance\_type = "t2.nano"

tags = {

Name = "test\_instance"

}

}

**💡 You might also like:**

* [Managing Multiple Terraform Environments Efficiently](https://spacelift.io/blog/terraform-environments)
* [5 Ways to Manage Terraform at Scale](https://spacelift.io/blog/5-ways-to-manage-terraform-at-scale)
* [How to Automate Terraform Deployments and Infrastructure Provisioning](https://spacelift.io/blog/terraform-automation)

**3. Configure GitHub Actions workflow for Terraform**

Let’s move onto the more interesting part: the GitHub Actions workflow definition. To define workflows to run on GitHub Actions runners based on events, create a [YAML file](https://spacelift.io/blog/yaml) inside the .github/workflows directory of the repository.

For our example, we define a .github/workflows/terraform.yaml file.

*.github/workflows/terraform.yml*

name: "Terraform Infrastructure Change Management Pipeline with GitHub Actions"

on:

push:

branches:

- main

paths:

- terraform/\*\*

pull\_request:

branches:

- main

paths:

- terraform/\*\*

env:

# verbosity setting for Terraform logs

TF\_LOG: INFO

# Credentials for deployment to AWS

AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

# S3 bucket for the Terraform state

BUCKET\_TF\_STATE: ${{ secrets.BUCKET\_TF\_STATE}}

jobs:

terraform:

name: "Terraform Infrastructure Change Management"

runs-on: ubuntu-latest

defaults:

run:

shell: bash

# We keep Terraform files in the terraform directory.

working-directory: ./terraform

steps:

- name: Checkout the repository to the runner

uses: actions/checkout@v2

- name: Setup Terraform with specified version on the runner

uses: hashicorp/setup-terraform@v2

with:

terraform\_version: 1.3.0

- name: Terraform init

id: init

run: terraform init -backend-config="bucket=$BUCKET\_TF\_STATE"

- name: Terraform format

id: fmt

run: terraform fmt -check

- name: Terraform validate

id: validate

run: terraform validate

- name: Terraform plan

id: plan

if: github.event\_name == 'pull\_request'

run: terraform plan -no-color -input=false

continue-on-error: true

- uses: actions/github-script@v6

if: github.event\_name == 'pull\_request'

env:

PLAN: "terraform\n${{ steps.plan.outputs.stdout }}"

with:

script: |

const output = `#### Terraform Format and Style 🖌\`${{ steps.fmt.outcome }}\`

#### Terraform Initialization ⚙️\`${{ steps.init.outcome }}\`

#### Terraform Validation 🤖\`${{ steps.validate.outcome }}\`

#### Terraform Plan 📖\`${{ steps.plan.outcome }}\`

<details><summary>Show Plan</summary>

\`\`\`\n

${process.env.PLAN}

\`\`\`

</details>

\*Pushed by: @${{ github.actor }}, Action: \`${{ github.event\_name }}\`\*`;

github.rest.issues.createComment({

issue\_number: context.issue.number,

owner: context.repo.owner,

repo: context.repo.repo,

body: output

})

- name: Terraform Plan Status

if: steps.plan.outcome == 'failure'

run: exit 1

- name: Terraform Apply

if: github.ref == 'refs/heads/main' && github.event\_name == 'push'

run: terraform apply -auto-approve -input=false

Let’s inspect the above **GitHub Actions workflow definition file** from top to bottom.

The first thing we should do in our workflow file is define the workflow name, but if you omit the name field in your workflow file, GitHub Actions will use the path of the workflow file relative to the root of the repository as the default name. This default name is usually the path of the file, including the directory and file name.

In our example, the name is set with the following line:

name: "Terraform Infrastructure Change Management Pipeline with GitHub Actions"

If we were to omit the name, in the configuration file, this would default to *.github/workflows/terraform.yaml*.

**Triggers**

Then, we define the triggers. This is achieved by setting different options for the on keyword. Here, we want to trigger the pipeline if there is a push or pull\_request targeting the main branch and any changes on the path terraform/\*\*.

This is done here:

on:

push:

branches:

- main

paths:

- terraform/**\*\***

pull\_request:

branches:

- main

paths:

- terraform/**\*\***

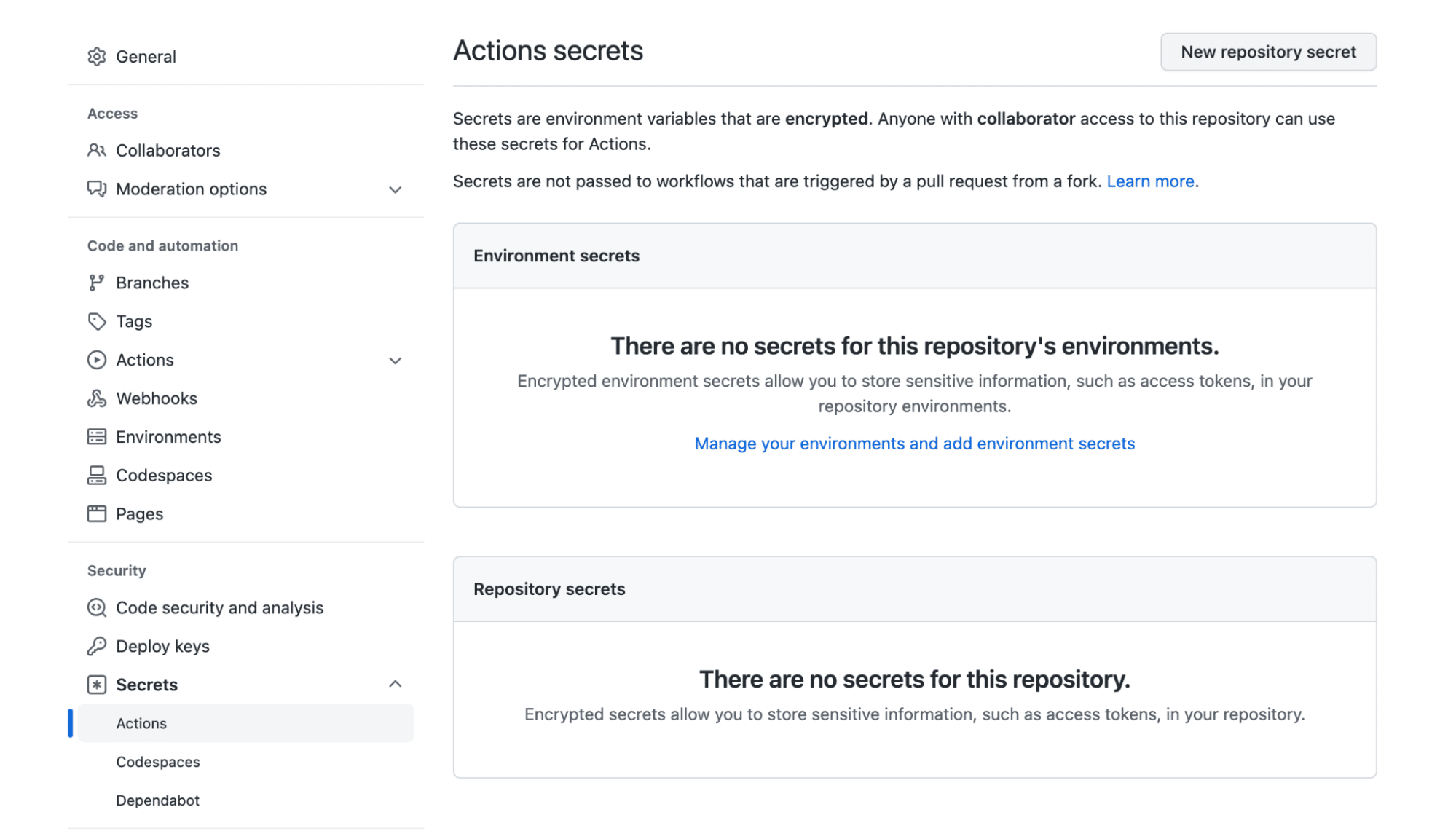
**Global environment variables**

Next, we define global environment variables that can be used at any pipeline step with the keyword [env.](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#env) These are the AWS credentials to deploy (because this is not a best practice for production, you should use short-lived credentials and assume an AWS role instead), the Terraform logs verbosity, and the bucket name that contains our Terraform state for this setup.

**Repository secrets**

Notice that we use the secrets functionality of GitHub repositories to store and fetch these credentials, and we don’t commit them to the repository.

If you wish to follow along, you can create these repository secrets from the Settings of your code repository by selecting New repository secret.



The env variables and the secrets are specified in this section:

env:

# verbosity setting for Terraform logs

TF\_LOG: INFO

# Credentials for deployment to AWS

AWS\_ACCESS\_KEY\_ID: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

AWS\_SECRET\_ACCESS\_KEY: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

# S3 bucket for the Terraform state

BUCKET\_TF\_STATE: ${{ secrets.BUCKET\_TF\_STATE}}

Keep in mind that secrets can be used whenever you want – you are not limited to using them only in the env section.

**Jobs — using GitHub Actions to run Terraform**

Continuing with decomposing the GitHub Actions file, we use the keyword [jobs](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobs) to define our single job terraform with the keyword [jobs.](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobs) We set the type of machine to run the job on with [runs-on](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idruns-on) and the default shell and working-directory with [defaults](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_iddefaults).

A job is composed of a sequence of steps. Each step is a task that runs in its own process in the GitHub Actions runner environment.

**1. Initialize the Terraform environment**

For our terraform job, we set up steps to checkout the repository to the runner and set up a specific [version of Terraform](https://spacelift.io/blog/terraform-version-upgrade). Then, we perform Terraform-related tasks as separate steps. We initialize our Terraform environment according to the backend configuration, check for correct formatting, and confirm the configuration is valid.

The checkout, Terraform installation, and initialization happen in this section:

- name: Checkout the repository to the runner

uses: actions/checkout@v2

- name: Setup Terraform with specified version on the runner

uses: hashicorp/setup-terraform@v2

with:

terraform\_version: 1.3.0

- name: Terraform init

id: init

run: terraform init -backend-config="bucket=$BUCKET\_TF\_STATE"

**2. Additional checks**

As a best practice, after Terraform init, we need to ensure that our Terraform code respects the standards we have set up in our organization, so in this example, we are checking if the code is formatted properly and the configuration is valid with this block:

- name: Terraform format

id: fmt

run: terraform fmt -check

- name: Terraform validate

id: validate

run: terraform validate

Of course, you could add other tools here that would help with your Terraform workflow before or even after the initialization. These include such as security vulnerability scanning tools like Tfscan, Checkov, Terrascan, Kicks, etc.

**3. Generate a plan**

For every pull request, we generate a Terraform plan for all the changes. To improve visibility, we also use a clever mechanism to comment on this plan on the pull request. This will ensure that our engineers see what resources will be created/changed/deleted without having to leave GitHub.

For a production-ready pipeline, we can then also include OPA policies that would parse the plan and can show warnings or even deny the plan if it is against the rules we have set up. As this is just a demo, we haven’t done this, but to create a more realistic example, we check if the plan had an error and force the pipeline to exit with a 1 exit code.

**4. Run terraform apply**

Finally, if we are happy with the changes and the Pull Request has been approved, we merge the changes to the “main” branch, which triggers the terraform apply command to perform the infrastructure changes.

Our example includes only a few of the many existing options and features for configuring workflows with GitHub Actions. Read the [official Workflow Syntax docs](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions) for more options and ideas of what is possible.

Setting a custom CI/CD pipeline with GitHub Actions to manage Terraform’s lifecycle and infrastructure provisioning is an excellent method for enforcing best practices, applying infrastructure changes predictably and safely, and eliminating the need for human intervention.

Read more about [how to manage GitHub with Terraform](https://spacelift.io/blog/terraform-github) and how to integrate [Pulumi with GitHub Actions](https://spacelift.io/blog/pulumi-github-actions" \t "_blank).

**Tips for troubleshooting GitHub Actions workflows**

Troubleshooting GitHub Actions Workflows can be cumbersome due to the complexity and variety of tasks they can perform. Here is a list of tips to identify and resolve issues:

1. **Check the logs** — This is your starting point for examining what happens in your GitHub actions workflow. Generally, GitHub Actions provides detailed logs for all the steps in your workflow. Identify the error messages and try to address them.
2. **Run scripts locally** — If your workflow uses any kind of scripts, try to run them locally if possible. This will speed up the debugging process and help you identify errors much faster.
3. **Use debugging tools** — Several tools are available for debugging your workflows. You can set ACTIONS\_STEP\_DEBUG secret to true, inside your repository to provide verbose logging.
4. **Check for syntax errors** — Ensure your workflow file is correctly formatted and free of errors. You can use yaml linters for this.
5. **Version pinning** — Pin actions and dependencies to specific versions to ensure that no breaking changes are added in minor/major updates.
6. **Examine env vars and secrets** — Make sure that all env vars and secrets are accurately set and can be accessed in the workflow.
7. **Temporarily simplify the workflow** — Remove some of the steps, until you identify the errors in your workflow and address them.
8. **Review dependencies** — Especially if you are not pinning versions, changes in some of the dependencies may break the current functionality of your workflow, so it is always useful to review the dependencies.

These are just a couple of tips to help with any issues with your GitHub Actions workflows. Depending on your workflow, some other steps can be taken to speed up the overall troubleshooting process.

**Terraform in GitHub Actions: Best Practices**

Using Terraform in GitHub Actions can be a powerful way to implement IaC in a CI/CD pipeline. Here are some best practices you should follow when you are defining the workflow:

1. **Use a specific Terraform version** — This ensures consistency across different runs and helps in avoiding unexpected behaviors due to upgrades.
2. **Take advantage of remote state** — State files can contain sensitive information, so use a secure storage such as AWS S3, Azure Blob Storage, or others to ensure your data is encrypted. Also, implement state locking to avoid conflicts. If you don’t use a remote state, your infrastructure will be created every time you run an apply, as your configuration won’t be aware that the infrastructure was created before.
3. **Use dynamic credentials** — The best way to authenticate to your cloud provider is by leveraging dynamic credentials, as they are short-lived and protect against breaches.
4. **Use environment variables for secrets if you are not using dynamic credentials** — This will ensure that at least you are not hardcoding sensitive information.
5. **Automate formatting and validation** — This will reduce the time and costs associated with your runs, as you will be able to identify errors before even running a Terraform plan.
6. **Integrate security vulnerability scanning tools** — In the long run, when you scale, you will add security vulnerabilities in your code without even noticing. That’s why security vulnerability scanning tools are really helpful. as you can see these vulnerabilities before deploying the actual code
7. **Create a plan → apply workflow** — It is imperative to see what’s going to change, before actually making the change, as you can make multiple mistakes with your code changes. Also, commenting the plan on a pull-request and ensuring that all engineers involved in the process review it thoroughly is crucial.
8. **Implement RBAC** — Ensure your runner has only the necessary permissions to run the tasks you want. Use RBAC on the cloud provider side to manage what actions Terraform can perform.
9. **Test your code** — Take advantage of Terratest or Terraform’s native test framework to ensure your infrastructure changes work as expected.
10. **Implement OPA policies** — Policies can help you with ensuring all your organization’s standards are met.

**Managing Terraform with CI/CD — scaling and operational concerns**

Setting up pipelines with a CI/CD tool like GitHub Actions for managing infrastructure with Terraform is a great approach and very popular, but this method requires substantial engineering efforts and considerable maintenance.

Firstly, running Terraform in CI/CD pipelines requires teams to take into account considerations for working with distributed systems with stateless components. Teams must take care of the state separately and consider using mechanics like version control and state locking to avoid disasters and race conditions.

Scaling and securing infrastructure management with Terraform and CI/CD pipelines in big organizations with [multiple environments](https://spacelift.io/blog/terraform-environments) and teams a major undertaking. Custom guardrails, policies, and controls must be integrated and set up in workflows. Handling dependencies and passing values between Terraform environments and workspaces require manual effort and possible extra tools. Gaining visibility on configuration drift requires additional tooling and processes.

To support an organization’s growing infrastructure needs, teams must invest time in continuously improving such setups and assume the overhead of maintaining custom-made solutions in the long term.

Read more about managing Terraform at scale in this article: [5 ways to manage Terraform at scale](https://spacelift.io/blog/5-ways-to-manage-terraform-at-scale).

**How Spacelift helps with managing Terraform at scale**