DevOps Self-Service Pipeline Architecture and Its 3–2–1 Rule

A high-level architectural overview of the self-service pipeline

DevOps Self-Service Calls for Drastic Mindset Change

Whether you call it DevOps self-service, democratizing DevOps, or Platform Engineering, the goal is the same — to empower developers with more access, control, and ownership over the pipelines to boost productivity.

Let’s call it “DevOps self-service” in this article to be consistent in our terms.

Why drastic mindset change? Let's start with exploring our self-service pipeline architecture from a high level.

Self-Service Pipeline Architecture

The diagram below depicts a generic self-service pipeline architecture based on a 3–2–1 rule (a term I coined while drawing the diagram): Three types of source code. Two types of pipelines. One pipeline integration glue. I mention the word "generic" mainly because there are many variations of the pipeline architecture based on the types of workloads. This generic pipeline design is more tailored for microservices architecture, not for serverless workloads. Serverless pipeline architecture should be even simpler.

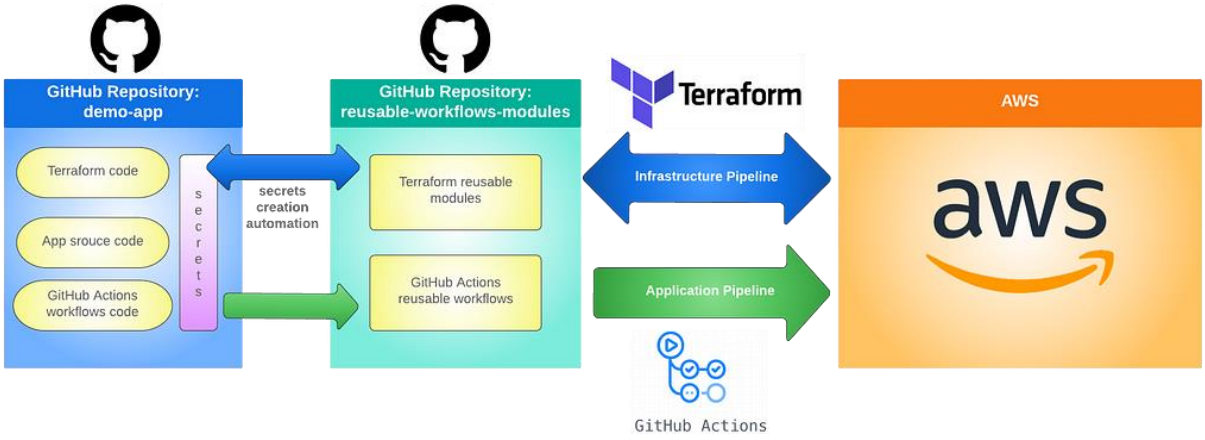

diagram by author

Let's dive in to see what exactly this 3–2–1 rule entails.

Three Types of Source Code

The following three types of source code will reside in the same GitHub repository for your microservice.

- Terraform code: yes, we have Terraform code residing in application repositories. This, perhaps, is the most challenging for some to accept as we have been used to having Terraform code centralized in a particular DevOps repo, no developers allowed. Now with DevOps self-service, we are relinquishing the Terraform code to developers, partially, into the application repos. Why partially? because we are keeping the Terraform reusable modules in a centralized repo still. More on Terraform reusable modules and Terraform project structure in my story titled DevOps Self-Service Centric Terraform Project Structure.

- Project source code: this is our good old source code for our app, be it Java, Node.js, Python, or other programming languages.

- GitHub Actions workflow code: for those who are familiar with GitHub Actions, this is no surprise, as the actions workflow yml files have to reside in the `.github/workflows` directory at the root of our app.

Two Types of Pipelines

- Infrastructure pipeline: IaC with Terraform. Picking Terraform here is mainly because of its cloud-agnostic and open source nature. A GitHub Actions Terraform workflow can kick off the infrastructure pipeline to provision cloud resources such as services in AWS.

- Application pipelines: our application CI/CD pipelines, kicked off by GitHub Actions workflows. These are the pipelines to build, test, scan, and deploy our apps into our cloud providers.

One Pipeline Integration Glue

- How do we tie infrastructure pipelines with application pipelines to make them work together seamlessly? We need a glue to integrate these two types of pipelines. And this glue is GitHub secrets creation automation. Upon successful infrastructure provisioning, we can use Terraform to automate GitHub secrets creation by calling the GitHub provider. Notice the double-ended arrows for the infrastructure pipeline in the diagram above, as the Terraform outputs for the secrets get inserted into GitHub automatically. This eliminates manual secrets creation and enables application pipelines to kick off CI/CD for the specified GitHub environment with the secrets associated with that particular GitHub environment provided by Terraform through the infrastructure pipeline. This accomplishes the end-to-end, state-of-the-art integration of these two types of pipelines.

High-Level Design of DevOps Pipelines

We will dive deeper into the project structures for Terraform, its reusable modules, and the GitHub Actions workflow orchestration in future stories. For now, let's take a high-level overview of what the two types of pipelines look like, what they do, and what detailed steps are involved.

Infrastructure Pipeline

Given its cloud-agnostic and open source nature, Terraform is the tool of our choice to build our infrastructure as code (IaC).

Terraform GitHub Actions workflow

Here is a high-level overview of what our Terraform GitHub Actions workflow looks like:

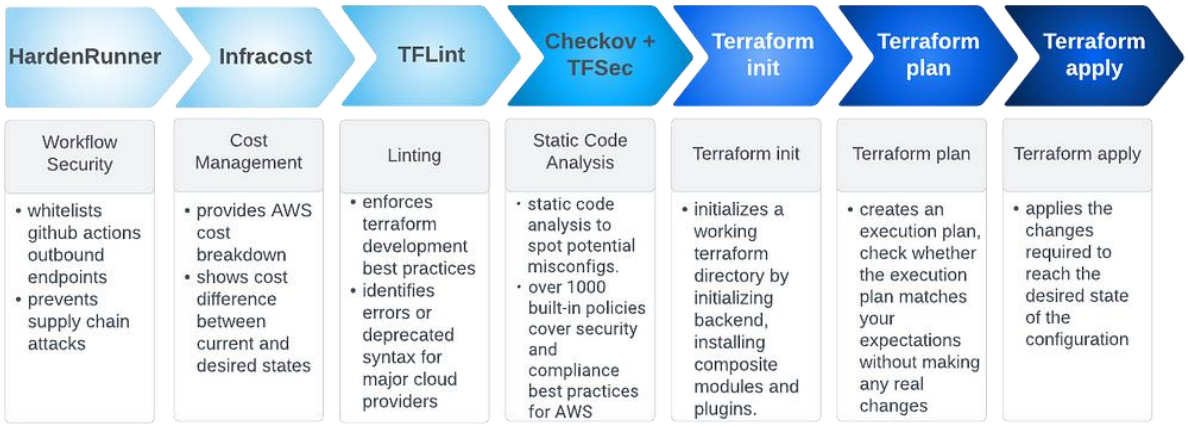

diagram by author

Note: The diagram does not depict alternative flows, such as for `terraform destroy`. You can always add alternative flows per your pipeline requirements.

Application Pipelines

Our application pipelines are developed using GitHub Actions. Below is a high-level overview of the two typical pipelines (CI and CD for microservices). These are mere examples. Your workflows could contain different steps depending on the nature of your applications.

Microservice CI GitHub Actions workflow

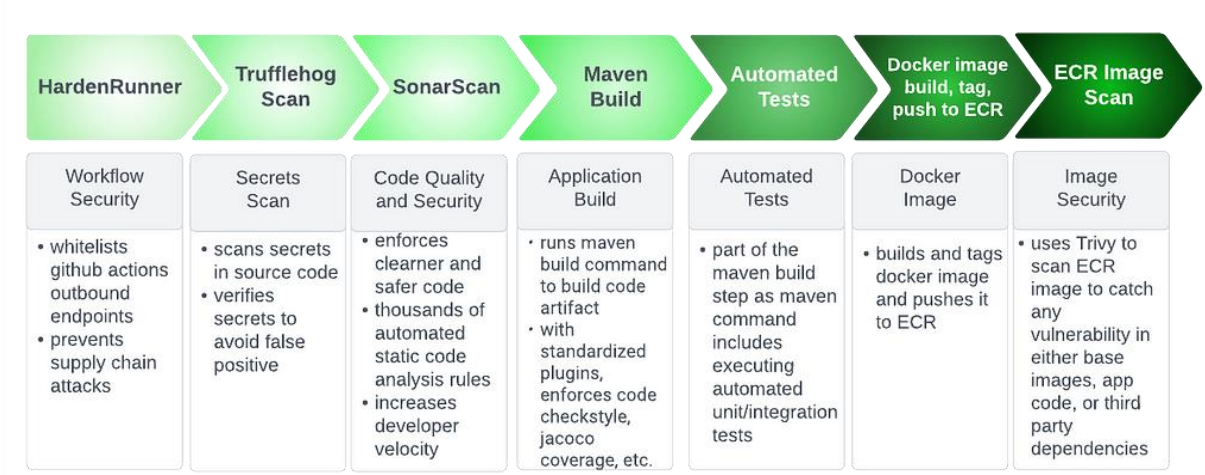

diagram by author

Microservice CD GitHub Actions workflow

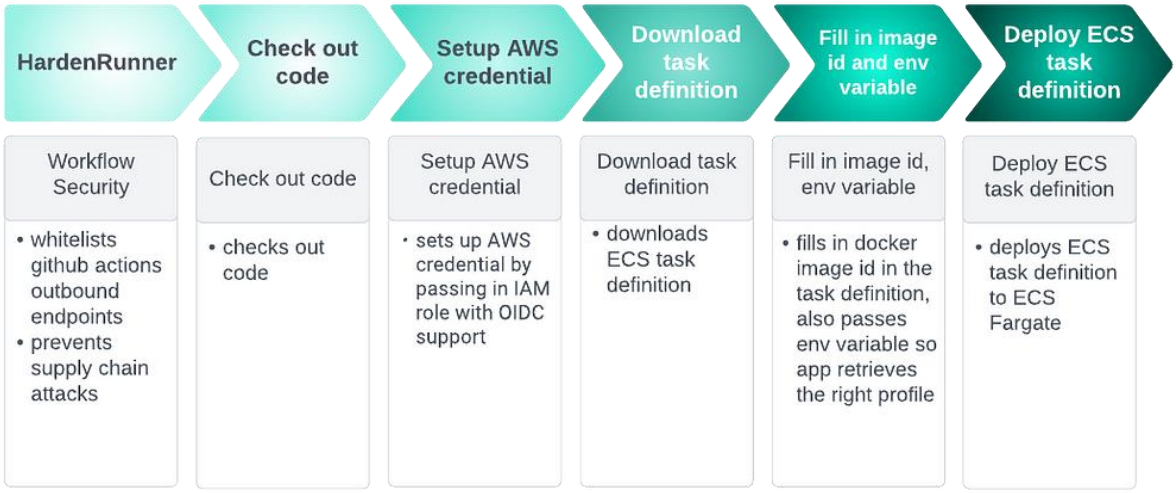

diagram by author

Infrastructure/Application Pipelines Integrated

Here's some key points to highlight:

- Terraform code and GitHub Actions workflow code reside in the application's GitHub repository.

- App's Terraform code calls Terraform reusable modules, which reside in a centralized GitHub repository.
- The infrastructure pipeline can be kicked off by either a manual trigger or PR creation/merge of Terraform code using GitHub Actions Terraform workflow to provision cloud resources.
- Upon successful infrastructure provisioning, Terraform automates GitHub secrets creation by calling the GitHub provider. Notice the double-ended arrows for the infrastructure pipeline in the diagram above as the Terraform outputs for the secrets get inserted into GitHub automatically. This eliminates manual secrets creation.
- Upon PR creation/merge of application code, or manual trigger, application pipelines are triggered by GitHub Actions workflows for CI and CD to build, test, scan, and deploy our app into the cloud.

Summary

DevOps self-service has gained traction in recent years mainly due to the growing trend of cloud-native architecture. This article focused on the overall self-service pipeline design and the main ingredients in DevOps self-service. The 3–2–1 rule outlines the overall architecture of a self-service DevOps practice. I hope you find this story helpful.

I welcome you to check out the rest of the four parts in my five-part "The Path to DevOps Self-Service" series:

DevOps Self-Service Centric Terraform Project Structure
How to structure Terraform code and its reusable modules
betterprogramming.pub

DevOps Self-Service Centric GitHub Actions' Workflow Orchestration
How to orchestrate GitHub Actions' workflows that are driven by image immutability
betterprogramming.pub

DevOps Self-Service Centric Pipeline Security and Guardrails
A list of hand-picked actions for security scans and guardrails for your pipelines, infrastructure, source code, base…
betterprogramming.pub

DevOps Self-Service Centric Pipeline Integration
Secrets management as the glue of pipeline integration
betterprogramming.pub

Happy coding!