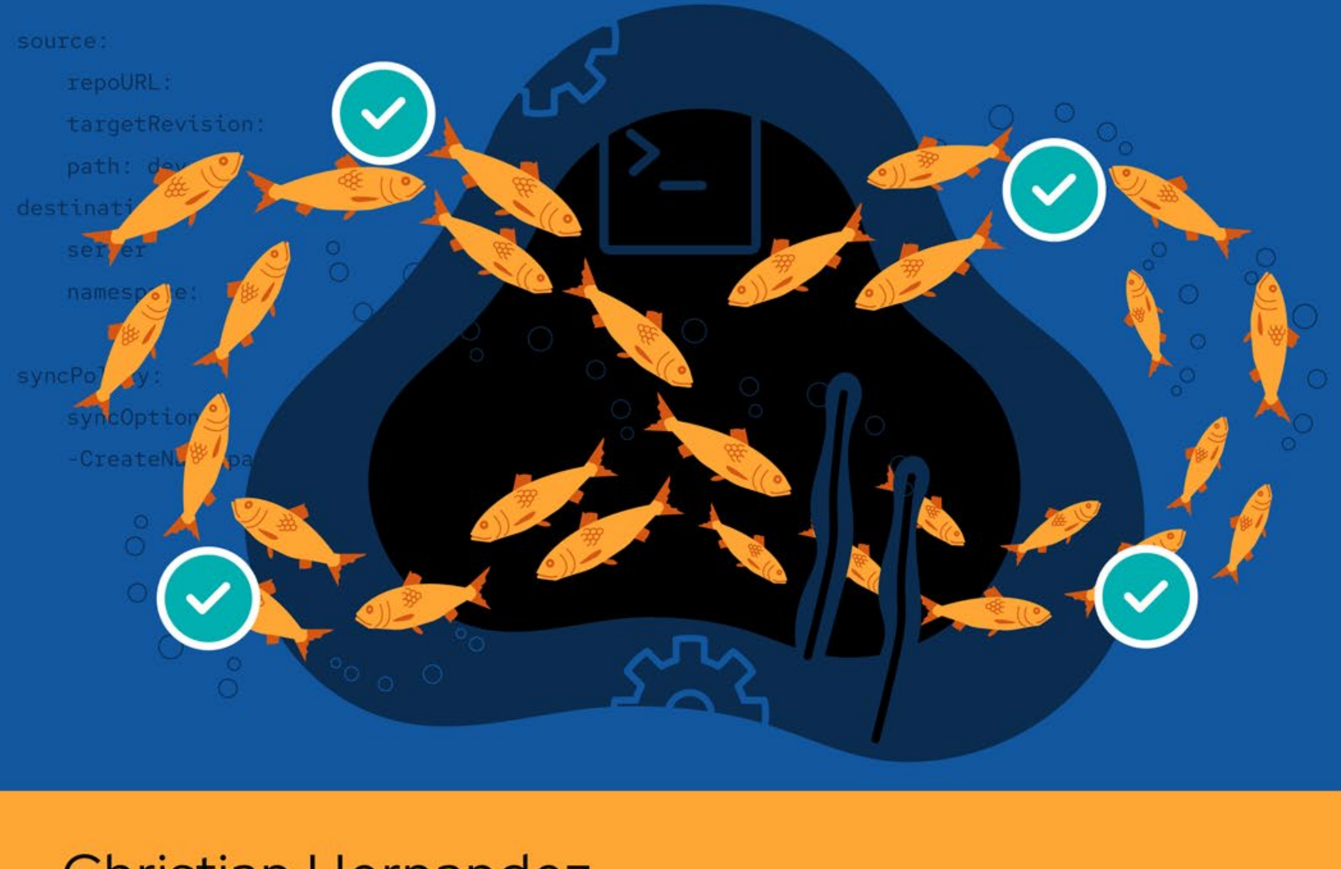
Published by Red Hat Developer

The Path to GitOps



Christian Hernandez

Foreword by Chris Short, Co-chair, OpenGitOps/GitOps Working Group

Contents

Contents

Contents

Foreword

In May 2018, I recorded a podcast [1] to talk about tools in the Kubernetes space. Specifically, continuous integration (CI) and continuous delivery (CD) tooling. My mindset at the time was, "We can do better." We had new, declarative tools for infrastructure and a new paradigm to work. However, there weren't many tools that had fully embraced Kubernetes natively. But a line in that podcast set the course for where I wanted things to head: "To me, GitOps is the holy grail of software and infrastructure management."

At the time, someone *was* trying to do better: A small, scrappy startup called Weaveworks. They're the ones who coined the term *GitOps* in 2017. Our futures would be entwined for a significant time after that podcast.

Fast forward to August 2019, and I was making an internal move at Red Hat from the Ansible team to the OpenShift (Red Hat's Kubernetes product) team. That was when I was introduced to Christian Hernandez. A font of Kubernetes knowledge, Christian would take the lead on building out some of the most complex systems our team could develop.

Not only that, he saw a lot of potential for Kubernetes and was looking for ways to extend it even further. In CustomResourceDefinitions (CRDs), Kubernetes users had a way to create all sorts of technology on Kubernetes clusters using familiar patterns. A more refined way to manage configurations and declare the state of everything in a given Kubernetes cluster called GitOps is something that Christian embraced wholeheartedly.

Fast forward again to 2020, and we struggled to adjust to life in a pandemic. Meeting customers one-by-one via the communication platform du jour wasn't scalable. Our team started live streaming, and on October 9, 2020, Christian and I hosted our first stream about GitOps [2]. It received very positive feedback and later became a recurring show. Eventually, this work led us to the Cloud Native Computing Foundation's GitOps Working Group (and later the OpenGitOps project).

By then, GitOps was a topic everyone wanted to know about. Suddenly, we were asked to spur discussions about GitOps methodologies and practices. We were asked to share our patterns for practicing GitOps, and to brain-dump the details into meetings, documents, glossaries, and more. We built the GitOps Principles [3] based on knowledge and feedback from a wide swath of industry experts. We organized multiple workshops and GitOpsCons and attended regular meetings to help shape GitOps.

The entire time, Christian Hernandez established himself as one of the preeminent experts in GitOps. He deftly fielded questions about the topic live on-air and answered them more often than not as if this were second nature. Christian embodies GitOps and applies it where it makes sense, so by following his implementations, you will adopt DevOps practices in the process.

Foreword

GitOps works both at a small level and on a colossal scale. With GitOps, your releases can measure in the hundreds per day. It quickens developer loops so much that often, features land in production that same day. All the golden metrics you track will improve, and morale along with it.

Christian has built out a lot of the foundation of GitOps. If you asked me for the best source of GitOps knowledge, I'd say you've found them already.

—Chris Short
Co-chair, OpenGitOps/GitOps Working Group
July 2022

References

[1] https://thenewstack.io/the-best-ci-cd-tool-for-kubernetes-doesnt-exist/
[2] https://youtu.be/UvwcVNv61Mo
[3] https://opengitops.dev/#principles

Introduction

As Christian was writing this book, I had the pleasure of learning alongside him, reviewing his practices, and testing them directly in my setups along the way. Examining the strategies he created while helping to mature our customer base, it was clear that Christian had developed a strong starting point for organizations looking to build their own GitOps practices. I'm really excited to see those practices summarized for the masses here in *The Path to GitOps*.

Toward the end of writing *The Path to GitOps*, Christian decided to pursue a new role outside of Red Hat. While we're sad to see him go, we're also celebrating the opportunity he's taken to continue to influence and evolve the GitOps community. We're excited to share Christian's enthusiasm for GitOps here in his first book, and we'll continue to support him and cheer him on as he continues to share his expertise across the GitOps community.

—Natale Vinto
Senior Principal Developer Advocate, Red Hat
Author, Modernizing Enterprise Java *and* GitOps Cookbook *(O'Reilly Media)*
July 2022

Chapter 1

What is GitOps?

GitOps has quickly become the tech industry's latest buzzword. Yet, when you search for GitOps, you will likely be presented with a lot of familiar concepts that seem unrelated. Is GitOps something that application developers use? Is it for infrastructure folks or system administrators? Is it something you can buy off the shelf? Or is it just a fancy new term for DevOps [1.1], or continuous integration/continuous deployment (CI/CD) [1.2]?

As a matter of fact, GitOps unifies a collection of different topics in automation, application delivery, infrastructure management, and security. In this chapter, I will go through each aspect of GitOps. By the end of the chapter, you will have a better understanding of what GitOps means, what it is, what it isn't, and more importantly, have it demystified for you.

Origins in DevOps

Discussions of GitOps naturally start with *DevOps*, a term on which GitOps is clearly based. Around 2007, many developers and site administrators began to identify gaps in application development processes. Although the DevOps movement officially started around 2007, I believe its true birth can be traced back to February 2001, when the Agile Manifesto [1.3] was published.

In many ways, the Agile Manifesto is the grandfather of DevOps. Agile development practices were a big step toward improving the application development process for the end user and for internal or external customers.

However, while Agile focused on customer and developer experience, the process of delivering that software was still stuck in old methodologies like waterfall. The process was speeding up, so the delivery of that software had to change to catch up.

The DevOps movement was born out of the need to automate application delivery. It allows the teams that wrote, delivered, and supported the software to work together in service of that goal. DevOps isn't necessarily a department, but rather a culture in your organization.

Kubernetes and containers

So how does GitOps fit with DevOps? That's simple: GitOps *is* DevOps. GitOps is the natural progression of DevOps, and it implements the best of what DevOps practitioners were already doing—they just didn't know it yet.

Containers [1.4] are the groundbreaking technology that has created the newest versions of DevOps. The Kubernetes [1.5] container platform, in particular, has fostered a whole new way of thinking about application deployment, because configuration files are used to declare the creation of container instances and dictate how the platform goes on to deploy or delete them. Thus, Kubernetes and containers created new challenges for DevOps, as well as new tools for carrying out the DevOps vision.

Weaveworks [1.6] is credited with pioneering the GitOps model. The story, described in a 2021 blog post [1.7], is very interesting. Back in 2017, Weaveworks was a Software-as-a-Service (SaaS) company that hosted applications on their platform using Kubernetes as the infrastructure layer managing the applications. One day there was an incident where a configuration change took down their entire hosting platform, but the DevOps engineers were able to bring back the system in about 40 minutes. When asked how they did it so quickly, they described their process, which Weaveworks CEO and cofounder Alexis Richardson called “GitOps.”

The DevOps engineers over at Weaveworks came up with a system that allowed them to return their entire platform (not just the workloads running on them) back to its original state. They essentially implemented *infrastructure as code* by keeping everything they wrote (including configurations) stored and versioned in Git, and by taking advantage of declarative configurations in Kubernetes.

Cloud-Native DevOps

So if GitOps isn’t anything new, why are we hearing so much about it now? Much like how the Agile Manifesto changed the game for developers and administrators, Kubernetes and other container technologies have changed the game for DevOps practitioners. The ways in which Kubernetes and cloud-native systems manage immutable instances (containers) have allowed DevOps practitioners to further refine their practices.

If DevOps is the culture, GitOps is the operating model that is best suited for cloud-native architectures. So, although GitOps isn’t necessarily new, it certainly feels that way because Kubernetes is still an evolving topic in the industry.

A DevOps Operating Model

Kubernetes uses a declarative model to define instances and automate their deployment. You can think about Kubernetes as a set of APIs that uses *state* as a central point of deciding what and how to operate on something. Kubernetes controls the operating environment by comparing the declared (desired) state of an object to the current running state. If these states differ, Kubernetes reconciles those differences.

To understand reconciliation, it’s useful to understand the terms *drift* and *autohealing*. If you declare that ten instances of your application should be running, but two fail in production, leaving only eight, your running state has drifted away from the desired state. If your DevOps process automatically detects the drift and starts two new instances, it is autohealing.

Kubernetes can achieve autohealing because containers are fungible and immutable. A container can be restarted or scaled at will, making it easy to manage workloads in this model.

It is important to grasp these Kubernetes concepts because GitOps is also based on this model. Thus, GitOps is the DevOps operating model that is used with Kubernetes and cloud-native architectures. Kubernetes lends itself to GitOps, and many GitOps principles are built on Kubernetes patterns.

GitOps Principles

In November 2020, Amazon, Codefresh, GitHub, Microsoft, and Weaveworks announced the creation of the GitOps Working Group. This working group was meant as a way for interested parties to get together and define what “GitOps” actually means, under the Application Delivery special interest group (SIG) in the Cloud Native Computing Foundation (CNCF) [1.8]. By the end of 2020, the GitOps Working Group was bootstrapped and became an official working group within the Application Delivery SIG.

In March 2021, the GitOps Working Group formed OpenGitOps [1.9] as a foundation for establishing interoperability between tools, conformance, and certification through standardized documents and code. OpenGitOps is currently a sandbox project within the CNCF.

In October 2021, the GitOps Working Group released the OpenGitOps Principles [1.10], a set of principles for managing software systems. We'll look at the four major principles in the sections that follow.

Declarative

The first OpenGitOps principle states:

> *A system managed by GitOps must have its desired state expressed declaratively.*

There are a few things to note here. First, a *system* in this context is defined as one or more runtime environments consisting of resources under management, the management agents within each runtime, and the policies for controlling access and management of repositories, deployments, and runtimes.

Second, note the reference to the *desired state*. This means that you represent the way you want the system to work in an "end state," which will be the final state achieved by changes made by the GitOps environment.

Lastly, the desired state must be *declarative*. The state of a system is stored as a set of declarations without procedures for how that state will be achieved.

Although you can store your declarations in any format, we will be focusing on YAML [1.11] for Kubernetes in this book.

Versioned and Immutable

The second OpenGitOps principle is:

> *Desired state is stored in a way that enforces immutability and versioning and that retains a complete version history.*

The canonical example of the "versioned and immutable" principle is Git, which is why it's the first element in the term GitOps. Git's store is versioned and immutable because each change is tracked in a new version without altering previous versions (except if the user requests a change explicitly, an extreme and rare event). So you can revert back to a previous version while preserving an audit of all the changes that have been made.

Although Git is a good example of this principle, you don't need to use Git as your state store. Anything that complies with this principle can be used in GitOps (S3 storage, for example). However, in this book we will be focusing on using Git as the state store.

Pulled Automatically

The third principle states:

> *Software agents automatically pull the desired state declarations from the source.*

This principle is where GitOps starts to differentiate itself from a traditional event-driven process (more on that in the next section).

Although triggering changes and updates via webhooks or other events is a valid way to automate builds, it's not GitOps. GitOps software agents (which I sometimes call *GitOps controllers*) check the desired state by pulling declarations from the state store at regular intervals, which means *polling* as well as pulling. In GitOps, there is no webhook that needs to be hit. Instead, there is a reconciliation loop. This design leads to the final principle.

Continuously Reconciled

The fourth and final principle is another way that GitOps differentiates itself from event-based workflows:

> *Software agents continuously observe actual system state and attempt to apply the desired state.*

This principle directly mirrors the functions of the Kubernetes controllers, but GitOps applies it to a whole application or infrastructure stack instead of just one object. We've seen that the desired state is pulled from configuration information that is versioned and stored in an immutable storage system. If there is a difference between the desired and running states, they are reconciled by changing the running state. And this is happening continuously at a regular interval. "Continuous," here, is understood in the industry to mean that reconciliation continues to happen at a chosen interval of time. Reconciliation doesn't have to be instantaneous.

Continuous reconciliation distinguishes GitOps from traditional CI/CD, where automation is generally driven by pre-set triggers. GitOps triggers reconciliation whenever there is a divergence. I explain where GitOps fits into CI/CD in the next section.

GitOps and CI/CD

CI/CD is one of the prominent practices in the DevOps movement. This practice delivers applications at frequent intervals to internal or external customers by automating stages of application development. The main concepts attributed to CI/CD are *continuous integration*, *continuous delivery*, and *continuous deployment*.

If GitOps is just an extension of DevOps, then where does GitOps fit into the CI/CD workflow that has traditionally been the cornerstone of DevOps practitioners? I'll try to answer that question in this section.

Traditional CI/CD Workflows

In a previous section, we introduced traditional CI/CD as being event-driven. This should be familiar to anyone who has been in the software industry.

As shown in Figure 1-1, traditional CI/CD is very linear, basing each stage on previous ones. That's why the term commonly used for the CI/CD build/test/deploy process is a *pipeline*. It can provide integration, delivery, and deployment in a continuous stream of releases.

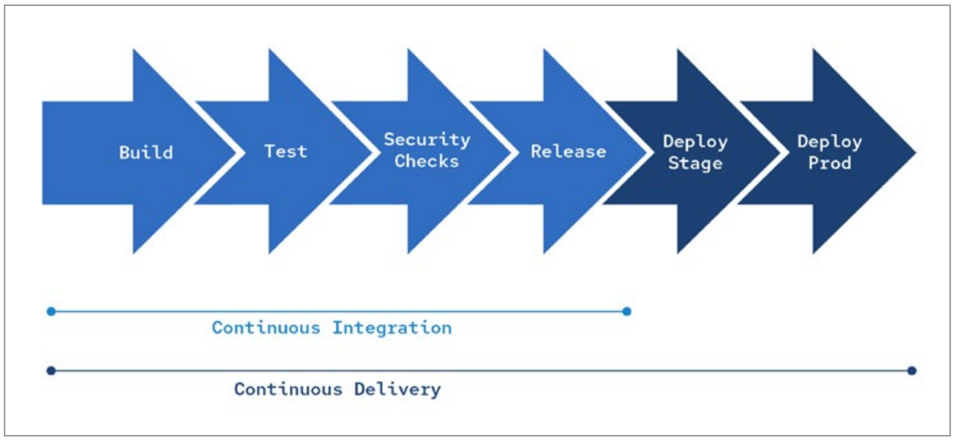


Figure 1-1: The CI/CD workflow.

Continuous integration builds and tests new code changes and merges them into a shared repository on a regular basis. CI is a form of rapid development that emerged as a result of the Agile Manifesto. CI is also a solution to the problem of having too many branches of an application, which might conflict with one another, in development at the same time. Commit early and often.

Continuous delivery refers to automating releases of changes to the dev/staging and pre-production environments. The changes can then, with the approval of the operation teams or release managers, get deployed to production. This aspect of CD is an answer to the problem of poor visibility and communication between development and business teams. It also automates the manual steps that slow down application delivery.

The purpose of continuous delivery is to deploy new code with minimal effort. Normally when people think of CI/CD, it stops at continuous delivery, where someone makes a commit and sees those changes relatively quickly in a dev/staging environment.

Continuous deployment takes continuous delivery one step further, deploying the changes into actual production (after they pass automated tests). Continuous deployment supersedes continuous delivery and is now seen as full DevOps automation.

Where does GitOps fit in?

GitOps doesn't replace CI/CD, but participates in it. While traditional tools like Jenkins and CloudBees focus on the entire CI/CD process, GitOps focuses on the CD aspect (both continuous deployment and continuous delivery). So now, instead of having deployment of your application be imperative in your CI/CD platform, it can be declarative based on a source of truth. Based on a reconciliation loop, the GitOps controller makes changes to the cluster by deploying new instances, once those changes have been committed to the state store.

How a deployment is done will be described in later chapters. But I can describe it here at a high level.

Under GitOps, the CI process just needs either to make a commit directly to your state store or to mark that the change is ready for deployment in a Git repository. In other words, if you're in GitHub [1.12], the recommended way to trigger deployment is by submitting a pull request; in GitLab [1.13] it's by submitting a merge request (the same idea with a different term).

Once merged, the change will be propagated by the GitOps controller. What's powerful about this method is that you no longer have to wait for the CI/CD process to finish before detecting and correcting drift. The GitOps controller acts as both an application delivery mechanism and a drift detection/autohealing process.

Operations via Pull Request

GitOps isn't just for developers; it's for administrators too. Since GitOps is a working model for DevOps on cloud-native infrastructures, everyone is (and should be) involved in the process. Administrators can take advantage of this automation, and those familiar with infrastructure as code will feel very comfortable in this new model.

The idea of GitOps is to apply the same workflow to any change that goes into infrastructure and applications. Changes are proposed by issuing a pull request to the respective Git repository. After reviews and approval, the change gets merged into the repository which is then applied to the target infrastructure. The complete history of the changes, review process, and deployment is visible through the Git history.

Modeling developer workflows for operations is a bit tricky, especially for those coming from more traditional working models. Although you're modeling infrastructure management after developer workflow, they aren't identical. Later on in this book, we'll go over how Git workflows for the infrastructure differ from the Git workflows from the application.

Summary

In this chapter, we introduced GitOps as a practice, took a quick look at its history, and showed its relationship to the DevOps movement and CI/CD. We then explored how Kubernetes was the catalyst for refining how we operate our infrastructure and application deployments today. We also took a look at the GitOps principles as defined by the OpenGitOps Sandbox project.

The next chapter introduces the basic tools for GitOps, notably Argo CD and Flux.

References

[1.1] https://developers.redhat.com/topics/devops

[1.2] https://developers.redhat.com/topics/ci-cd

[1.3] https://agilemanifesto.org/history.html

[1.4] https://developers.redhat.com/topics/containers

[1.5] https://developers.redhat.com/topics/kubernetes

[1.6] https://www.weave.works

[1.7] https://www.weave.works/blog/the-history-of-gitops

[1.8] https://www.cncf.io

[1.9] https://opengitops.dev

[1.10] https://github.com/open-gitops/documents/releases/tag/v1.0.0

[1.11] https://www.redhat.com/en/topics/automation/what-is-yaml

[1.12] https://github.com

[1.13] https://gitlab.com

Chapter 2

Tools of the Trade

In this chapter, we will go over the various tools used most often to manage GitOps workflows. Many of the tools were developed to fill a gap in earlier tools used in Infrastructure as code (IaC). Therefore, we will go over a brief history of Infrastructure as code, then examine the tools that arose from the IaC paradigm. We will also discuss the most popular GitOps tools, what they have in common with each other, and the benefits they offer.

Infrastructure as Code

Infrastructure as code is a method for managing, configuring, and updating the entire software infrastructure at a datacenter using configuration files that are both machine-readable and human-readable. The configuration files are machine-readable in a structured format (usually YAML) so that software tools can read them to automate changes. They are also human-readable so that administrators can maintain them easily and immediately understand what they define. You can think of configuration files as the "code" for the infrastructure.

Thus, Infrastructure as code lets administrators leave behind manual processes and avoid the drudgery of entering fields into a web form. Instead, they use configuration files to manage every component of the datacenter, from bare metal servers to virtual machines to networking equipment.

Automation reduces workload and prevents casual errors. However, its main benefit is scalability; the process can manage thousands of nodes and dozens of datacenters.

History of Infrastructure as Code

Infrastructure as code grew from the need to scale up quickly in the cloud. Amazon Web Services launched its Elastic Compute Cloud (EC2) in 2006, letting clients scale their infrastructure on demand. EC2 soon became popular, creating a new widespread problem: Administrators couldn't scale their application instances to take advantage of the seemingly unlimited amount of compute resources provided by cloud computing. The old methods of managing and configuring infrastructure wouldn't work at a massive scale.

The advantages of Infrastructure as code are cost and speed. The cost aspect can be broken down into staff capacity and the amount of time spent configuring and managing systems. The concept "do more with less" applies here. With IaC, you no longer need to hire at a "person per node" ratio. Removing the need for manual configuration frees up administrators to do other things.

IaC gives you vastly more speed because you can manage thousands of nodes all at once. You also reduce the risk that could be introduced by human error, because all the nodes get the same configuration.

As IaC grew in popularity, it helped drive the rise of the DevOps movement. Many new tools emerged as a result, including Chef, Ansible, Terraform, and CFEngine. One of the most popular tools was Puppet, which provided a way to configure systems in a platform-agnostic way. Any variations in the platform could be represented in Puppet as structured fields with variables that could take on different values for different operating systems, types of servers, etc. Thus, Puppet adopted a declarative model that we saw in the previous chapter as being central to Kubernetes and GitOps. The declarations were much easier to maintain than specifying the actual commands to run, which is *imperative* (although both models were still possible). .

Challenges of Infrastructure as Code

With all the popularity and usefulness of IaC, it does come with some drawbacks. There is definitely overhead to managing your system as code. Sometimes there are thousands upon thousands of lines of code that you need to go through, so issues aren't immediately obvious. Also, one mistake can have a huge impact on your infrastructure, and rollback can take some time to propagate.

IaC adds convergence time as well. If there's a drift in your system, it won't get fixed until the next time you run your IaC tool. You can set the tool to run at regular intervals or when your workflow triggers a change. But the idea behind IaC is to make it mostly event-driven. In other words, changes to your platform should adapt to changes automatically instead of requiring human intervention.

There is also the issue of *idempotency*, which refers to a change that can be applied over and over again with identical results. For instance, a command that updates a field in a database to a fixed value can be issued multiple times safely. Doing so might be wasteful, but the result is the same each time, so the operation is idempotent. In contrast, adding a line to a log file leaves multiple lines for the event, so it is not idempotent.

This issue doesn't seem obvious at first, because the system can usually rerun the commands that carry out the declaration. But suppose the commands try to update a local database on a system where a schema change had occurred, or after an administrator had deleted the whole database. In those scenarios, the command will fail.

So, the mutability of servers and virtual machines is the biggest challenge to IaC. This is where containers provide an advantage.

Containers Change the Game

Containers represent a totally different view of continuity in your infrastructure. You make software changes not by updating an existing configuration, but by deleting the whole virtual system and creating a new container. So in the example of updating a database in the previous section, Kubernetes heals the system simply by starting a new set of containers. That's it.

To get the same behavior from IaC and fixed servers, the IaC tool would have to destroy the node and recreate it every time there is immutable drift, which isn't too practical. This is the advantage that Kubernetes gives you.

Argo CD

The first GitOps tool we'll discuss is Argo CD [2.1]. Argo CD was written with GitOps in mind, to deliver changes to a Kubernetes cluster at massive scale. It detects and prevents drift in your Kubernetes clusters by working with raw YAML stored in a Git repository and using the apply functionality in Kubernetes.

Argo CD can also work with Helm [2.2] and Kustomize [2.3] to render the YAML produced by those tools before applying them to the Kubernetes cluster. (We'll look at Helm and Kustomize more closely in upcoming chapters.)

Argo CD is part of a larger ecosystem called the Argo Project [2.4]. It maintains a suite of tools that also includes Argo Workflows, Argo Rollouts, and Argo Events. Argo Labs serves as an incubator for other tools related to Argo, which can be used together or standalone.

Argo CD is one of many tools under the Argo Project umbrella that came from Intuit's acquisition of Applatixt [2.5]. Argo CD was created to fill a need in the company's infrastructure. Later, Intuit decided to release the Argo Project toolsets as open source. Since then, many companies, including Akuity, BlackRock, Codefresh, and Red Hat, have become part of the Argo community as active maintainers. Argo CD currently has incubating status in the CNCF.

Flux

In the chapter *What is GitOps?* I told the story of how Weaveworks recovered from an outage, inspiring cofounder Alexis Richardson to coin the term "GitOps." Flux (and Flagger) were tools that emerged from that environment to help people on their GitOps adoption journey.

Flux performs many of the same functions as Argo CD, but it is different in a lot of ways. The main difference is that Flux uses the Helm Golang [2.6] package. Flux doesn't render the YAML, but instead deals with Helm directly. This design gives users a familiar feel if they are already using Helm often.

Originally, Flux was built as a monolithic "do it all" operator modeled after what Weaveworks felt that it meant to do a GitOps workflow. With Flux version 2, the functions were broken up into individual components, called the GitOps Toolkit, and were based on controllers. These components include the source controller, the Kustomize controller, the Helm controller, the notification controller, and the image automation controller.

Flux was born out of the best practices and use cases from site reliability engineers (SREs) over the years at Weaveworks. It is now widely adopted by many companies [2.7] and continues to grow as the popularity of GitOps grows. Flux currently has incubating status in the CNCF.

Open Cluster Management

Open Cluster Management (OCM) [2.8] is the upstream project used by Red Hat Advanced Cluster Management for Kubernetes [2.9]. OCM has its roots in IBM Multicloud Manager. The tool goes beyond just applying YAML and detecting drift: It also manages the lifecycle of your Kubernetes cluster. The idea behind OCM and Red Hat Advanced Cluster Management is multicluster management from a single pane of glass. As with other GitOps tools, you can use these to keep your cluster in sync with a Git repository and detect and fix drift. But additionally, you can also manage the lifecycle of those clusters. From creation to destruction and policy enforcement, OCM and Red Hat Advanced Cluster Management take the idea of GitOps a step further and manage clusters with IaC as well.

Other GitOps Tools

It's worth mentioning several other tools that have emerged as the popularity of GitOps has grown. Some of these are in the early stages of development and others are solving a very specific problem.

PipeCD

PipeCD [2.10]lets you manage and promote deployments from one environment to another. Administrators use *control planes* to manage different deployments on one cluster or many clusters, each one running a PipeCD agent. This control plane is stateless. Its central task is to manage deployments on other clusters running the PipeCD agents. The control plane manages authentication as well.

Keptn

Developed by Dynatrace, Keptn [2.11] was designed to provide visibility into your environments. It aims to provide Service Level Objectives (SLOs) throughout the multi-stage deployment–for example, response time must not exceed 200ms during peak load). Keptn is meant to automatically set up environments (such as dev/staging/prod) with gating based on metrics for your CI/CD workflows. You can also integrate testing tools, do performance testing, chaos engineering, and more.

Pulumi Kubernetes Operator

The Pulumi Kubernetes Operator [2.12] is particularly interesting because it broadens the Kubernetes idea of declarations to make them more appealing to developers. Usually, when administrators think of declarations within the context of Kubernetes, they assume that the source code is YAML. Yes, the source of truth is stored in Git, and the declarations are usually in YAML, but it doesn't have to be. The Pulumi Kubernetes Operator lets developers write declarations in their chosen language: Golang, TypeScript, Python, etc. Instead of translating their infrastructure ideas into YAML, developers can write the declarations in the language they use for the rest of the project.

Summary

In this chapter, we reviewed the origins of Infrastructure as code, along with its advantages and shortcomings. We also saw how Kubernetes changed the game for Infrastructure as code. We went through popular GitOps tools, what they have to offer, and what choices you have when looking for a GitOps controller.

In this book, we will focus on Argo CD and Red Hat Advanced Cluster Management for Kubernetes. However, this book is meant to be agnostic, so almost everything you will read about can be applied to other tools.

References

[2.1] https://argoproj.github.io/cd/
[2.2] https://helm.sh
[2.3] https://kustomize.io
[2.4] https://argoproj.github.io/
[2.5] https://blog.argoproj.io/applatix-joins-intuit-7ab587270573
[2.6] https://go.dev
[2.7] https://fluxcd.io/adopters/
[2.8] https://open-cluster-management.io/
[2.9] https://cloud.redhat.com/products/advanced-cluster-management
[2.10] https://pipecd.dev
[2.11] https://keptn.sh
[2.12] https://www.pulumi.com/docs/guides/continuous-delivery/pulumi-kubernetes-operator/

Chapter 3

Templating

When you first start out with GitOps workflows, you'll notice something bothersome: you're dealing with a *lot* of YAML. (Some folks use JSON, but the same observation applies). In this chapter, we will see how to minimize the management of these lengthy configuration files and go over some of the best available practices and tools. Kustomize, Helm, and Kubernetes Operators all have roles to play for some environments.

Everything in Git

The GitOps Principles [3.1] (specifically #1 and #2) declare that the system's state is immutable and is described declaratively. In a Kubernetes world, this means that YAML manifests are stored inside Git. So the question often becomes, "How do I declaratively describe my resources in Git without copying and pasting the same YAML everywhere?"

It might seem like you'll have to duplicate a lot of the same YAML after you consider things like environments, clusters, regulatory restrictions, and anything else in your organization that might force you to create a lot of YAML with only slight variations between files. Luckily, there are tools that help you minimize this grunt work and help you avoid duplicating manifests.

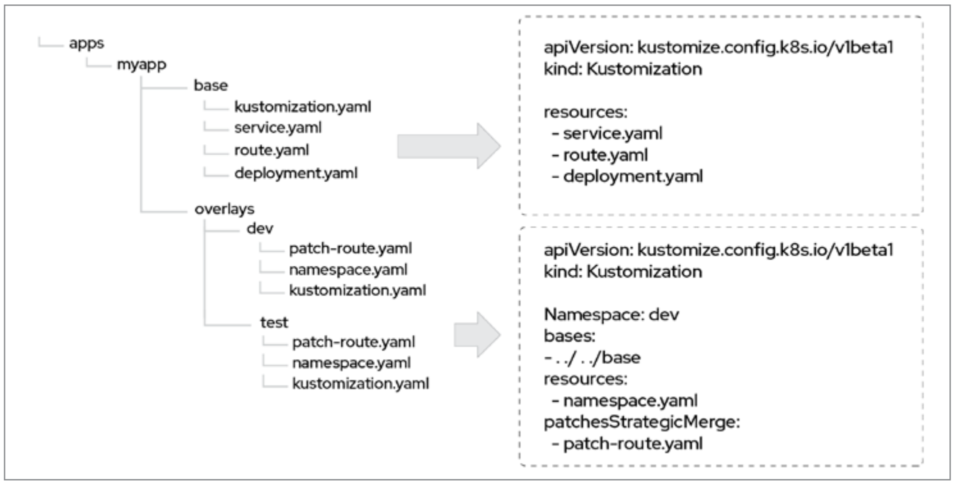
Kustomize

Kustomize is a framework for patching–or selectively altering–files, built into Kubernetes. It is a configuation manager that lets you customize untemplated YAML files without touching the original YAML configuration file.

Kustomize is organized in a hierarchical directory structure of *bases* and *overlays*. A base is a directory with a `kustomization.yaml` file containing a set of resources and associated customizations. A base has no knowledge of an overlay and can be used in multiple overlays.

An overlay is a directory with a `kustomization.yaml` file that refers to other Kustomize directories as its bases. An overlay can draw from multiple bases. It composes all resources from bases and can also add customizations on top of them. You can also write `kustomization.yaml` files that build on one another (for example, a base can refer to another base).

Figure 3-1 shows a typical use for Kustomize. A directory structure containing many files is shown on the left, and excerpts from particular `kustomization.yaml` files on the right.



```
└── apps
    └── myapp
        ├── base
        │   ├── kustomization.yaml
        │   ├── service.yaml
        │   ├── route.yaml
        │   └── deployment.yaml
        └── overlays
            ├── dev
            │   ├── patch-route.yaml
            │   ├── namespace.yaml
            │   └── kustomization.yaml
            └── test
                ├── patch-route.yaml
                ├── namespace.yaml
                └── kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
  - service.yaml
  - route.yaml
  - deployment.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

Namespace: dev
bases:
- ../../base
resources:
  - namespace.yaml
patchesStrategicMerge:
  - patch-route.yaml
```

Figure 3-1: Kustomize example.

In Figure 3-1, the `base` directory in the directory structure on the left contains the `kustomization.yaml` file that is the foundation for the others. This file is made up of a Deployment, a Service, and a Route (which could be replaced by an Ingress if you're using an ingress controller). These manifests are generally the same, and there are only slight differences between them in the `dev` and `test` environments.

Now let's do some customization. To render the YAML output for the `dev` environment, run:

```
$ kubectl apply -k apps/myapp/overlays/dev
```

The preceding command takes the base YAML and overlays what's in the `dev` directory, applying the resulting YAML into a Kubernetes cluster.

Kustomize validates YAML before deploying it and is agnostic regarding which GitOps tools you choose (Argo CD, Flux [3.2], Anthos Config Management [3.3], etc.).

Kustomize is powerful because it eliminates needless duplication of YAML and enables reuse through customization (patching). This means that you can store differences as deltas instead of copying the YAML in multiple places. The hierarchical structure provides flexibility by creating a series of overlays that can leverage other bases and other overlays, cascading over a sequence of files. Those overlays can refer to remote repositories as well.

Kustomize is included with Kubernetes, so there's nothing else to install. This is the basis of most structures and examples in this chapter.

Helm

Helm has become the de facto package manager for Kubernetes. If you've worked on a Kubernetes cluster previously, there is a good chance that you have used Helm at some point for its automation benefits. Helm provides not only a method of packaging an application and parameterizing YAML manifests but also a templating engine that can deploy your application to different environments.

Helm consists of *charts*, which are packaged and templatized versions of your YAML manifests. You can inject values into the parameters defined in the templates, and Helm injects these values into the manifests to create a *release*. A release is the end-state representation of the YAML that is deployed to your Kubernetes cluster. The information is stored as a secret on the Kubernetes cluster.

Figure 3-2 depicts each of these components.

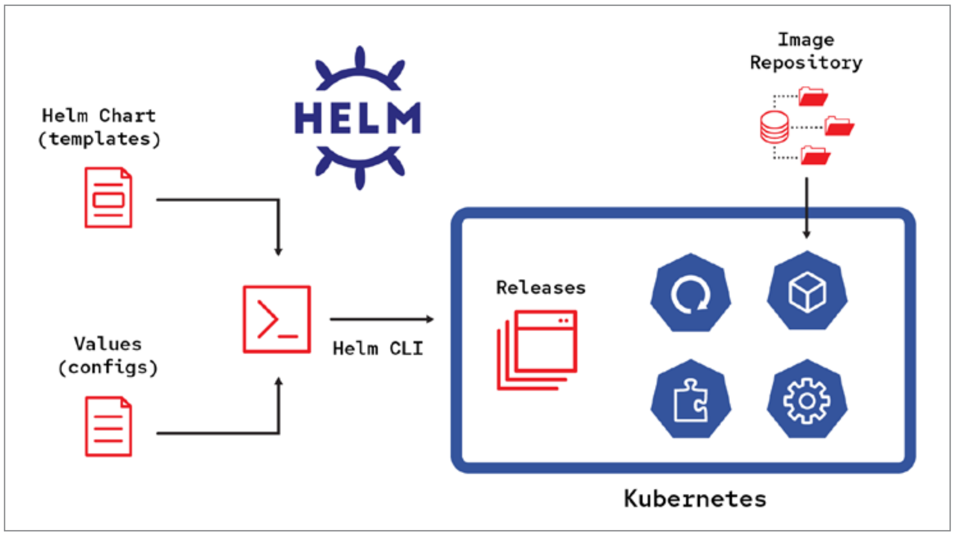


Figure 3-2: Helm overview.

Helm has a large ecosystem and many repositories that end users can draw on to de-ploy pre-built applications.

If your organization uses Helm heavily, you're in luck! Most GitOps tools support de-ploying Helm charts.

Operators

Operators aren't exactly a tool to help avoid duplicating YAML, but they fit into this chapter because they definitely can help. So what exactly are Operators?

Operators are built on top of Custom Resource Definitions (CRDs) [3.4] in Kubernetes. A CRD lets you define a resource Kubernetes can manage that is not part of the core primitives that Kubernetes is used to working with. For example, Kubernetes knows how to work with pods, but what if you have resources in your environment that you want Kubernetes to manage, such as virtual machines, databases, or load balancers? You can tell Kubernetes to manage such resources with CRDs.

CRDs sometimes come under strain by the complexity of such resources. Imagine you're managing a database and you want Kubernetes to take care of backup, restore, scaling, and schema changes–basically, run the resource on autopilot. This is where Operators come in. Operators codify operational knowledge in a way that Kubernetes understands.

Where do Operators fit in a GitOps world? And in the context of this chapter, where do they fit with templating and eliminating the duplication of YAML? Because specific tools exist for writing Operators (like the Operator SDK and KubeBuild), you can effectively write the logic for how your application gets deployed. You can write the Operator with this in mind and add the GitOps logic to your application. You can also use Kustomize and Helm alongside your Operator.

Combining Tools

With all these options, what tool should you use? It's not a question of "Kustomize versus Helm," but rather "Kustomize *and* Helm." Using Kustomize and Helm is a "yes and..." and not an "or" conversation. Most of the time, you will use them in tandem. Take a look at the Kustomized Helm [3.5] example from the Argo Project.

If you are using mainly raw Kubernetes manifest files, your best bet is to use Kustomize as much as possible. It's not only built into Kubernetes, but it's built into many GitOps tools as well.

Although you can do a lot with Kustomize, it's not technically a templating engine–it's a patching framework. If you need to parameterize certain things or come from the Helm world, you might think about either leaning all into Helm or starting to write Helm charts.

Helm is valuable when you want to parameterize your configurations instead of patching them. You usually parameterize when you don't know something about the cluster ahead of time (for example, the Ingress "host" field in the YAML). You can use Helm to parameterize the configuration and just supply the values specific for that deployment.

While Kustomize was built as a patching framework, Helm was built from the ground up as a templating engine. It also acts as a package manager, like DNF or `apt-get`. This means you can start treating Kubernetes as your "cloud operating system" and have Helm install your application as if it were packaged. Also, most GitOps tools support both Helm and Kustomize.

Operators can help with snowflake situations when you need more logic put in place beyond just applying a manifest. Operators can help by putting some declarative language around your imperative actions. Also, you can simplify what you are storing in Git by writing a lot of that complexity into the Operator. Since Operators are built on top of Kubernetes CRDs, all GitOps tools work with them.

Keep in mind that you are storing the GitOps controller configs in Git as well. The GitOps configurations should also be in Git and can be Kustomized, Helm-ized, or be turned into an Operator. For example, you can use Argo CD to manage itself [3.6].

Summary

In this chapter, we went over some of the options at your disposal for templating and patching your GitOps manifests. We went over how you can use Kustomize to natively create overlays that only change the deltas between your environments, how Helm can be used as a templating engine for your GitOps manifests, and how Operators fit into the picture. The next chapter will cover how to structure your Git workflows to best take advantage of GitOps.

References

[3.1] https://github.com/open-gitops/documents/blob/release-v1.0.0/PRINCIPLES.md

[3.2] https://fluxcd.io

[3.3] https://cloud.google.com/anthos/config-management

[3.4] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[3.5] https://github.com/argoproj/argocd-example-apps/blob/master/plugins/kustomized-helm/README.md

[3.6] https://argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/#manage-argo-cd-using-argo-cd

Chapter 4

Git Workflows

Your Git workflows are at the center of your GitOps deployments because workflows are the means of implementing your changes in your environment. When you adopt GitOps, Git is not only your source of truth (as it is for most projects) but also your interface into your environment. Developers have used Git workflows for their application delivery method for years, and now operations teams will have to adopt similar workflows.

But there are key differences between how you manage your code in Git and how you manage your GitOps configuration in Git.

In this chapter, I go over these differences and describe the best practices you should follow to make the best use of Git workflows for your GitOps deployments. We will see how to separate your configuration from your code, how to use branches, how to use trunk-based development workflows effectively, and tips for setting up policies and security for your Git workflows.

Separate Your Repositories

There are a few things to keep in mind when setting up your Git workflows for your GitOps directory structure and GitOps in general. The first is to keep your application code in a separate repository from your YAML configurations. This might seem counterintuitive initially, but most teams that start with code and configurations together quickly learn it's better to separate them.

There are a few reasons for separate repositories. First, you don't want a configuration change (such as changing the scale of a deployment from three to four nodes) to trigger a rebuild of your application if your application code didn't change. Another reason is that the approval process of getting a change into an environment shouldn't hold back continuous integration of your code. In general, application code and configuration information have independent lifecycles.

Also, many organizations separate the deployment process into several different teams. A lot of the time, the operations or release management team takes care of the application's release. Although DevOps aims to reduce barriers between teams and their activities, you don't want one team's process to slow down another.

Separate Development in Directories, Not Branches

Another best practice that surprises many programmers is to separate environments–such as test and production–into different directories, but not create branches for them. Like the separation of code and configurations, this principle might seem to go against the grain of version control, but keeping track of environmental branches can be a challenge.

One of the difficulties you might encounter if you manage workflows through branches is that promotion from one environment to another isn't as simple as a merge. You can see this issue with a simple example of updating an image tag. The application has been built, tested, and deemed ready to go from a sandbox environment to a test environment. But updating the image tag comes with other changes you don't want to merge. What about the scale in the Deployment [4.1]? What about the ConfigMaps and

Secrets? Those are bound to change in different environments and include things that should not be merged into other environments.

In short, every environment has configuration details specific to that environment. You can manually make the changes one by one or "cherry-pick" [4.2], but then your "simple merge" is no longer that simple. When you're constantly cherry-picking or making manual changes, the effort level outweighs the benefits of trying to mirror the application workflows.

Another danger of using branches and cherry-picking your way into production is that this will likely introduce a significant drift. As you get further along in the life of a software project, when it spawns hundreds of environments with dozens upon dozens of applications, you can quickly see how cherry-picking and making manual changes can get out of hand. You can no longer use a simple diff [4.3] to see the differences between branches, as the differences will be astronomical.

In the world of Kubernetes, the Kustomize [4.4] patching framework, and the Helm [4.5] package manager, using branches for environments is an antipattern. Kustomize and Helm make using directories and overlays for your environments easier. Kustomize, in particular, allows you to have a core set of manifests (called a "base" in Kustomize) and store the deltas in directories (called "overlays" in Kustomize). You use these overlays as directories with specific environment configurations in these directories.

Note: I will discuss Kustomize in detail in another chapter.

So do you use branches at all? Yes, but not in the way you think. With GitOps, trunk-based development has emerged as the development model for your configuration repositories.

Trunk-Based Development

The recommended workflow for implementing GitOps with Kubernetes manifests is known as trunk-based development [4.6]. This method defines one branch as the "trunk" and carries out development on each environment in a different short-lived branch. When development is complete for that environment, the developer creates a pull request for the branch to the trunk. Developers can also create a fork to work on an environment, and then create a branch to merge the fork into the trunk.

Once the proper approvals are done, the pull request (or the branch from the fork) gets merged into the trunk. The branch for that feature is deleted, keeping your branches to a minimum. Trunk-based development trades branches for directories.

You can think of the trunk as a "main" or primary branch. `production` and `prod` are popular names for the trunk branch.

Trunk-based development came about to enable continuous integration and continuous delivery by supplying a development model focused on the fast delivery of changes to applications. But this model also works for GitOps repositories because it keeps things simple and more in tune with how Kustomize and Helm work. When you record deltas between environments, you can clearly see what changes will be merged into the trunk. You won't have to cherry-pick nearly as often, and you'll have the confidence that what is in your Git repository is what is actually going into your environment. This is what you want in a GitOps workflow.

Policies and Security

Part of the challenge with trunk-based development is that now there is a single branch where things can go wrong. When relying on Git as your source of truth, it can be quite scary to depend on a single branch for not only your production environment but your organization as a whole. So you need to pay special attention to the features that version control offers for policy management and security to protect your trunk and provide stability to your environment.

When setting up your Git repository policies, use GitHub's branch protection rules [4.7] (or the equivalent from other Git providers). Setting branch protection rules provides several benefits, the most important of which is preventing someone from force pushing a change into the trunk (which in turn makes an immediate alteration to your environment). Branch protection also protects the branch from being accidentally or intentionally deleted. There are other advantages to protected branches, but the main takeaway is this: You need to trust what is in Git because it is in charge of managing your environment. Take every precaution that builds trust.

Also, set up rules as to who can perform a merge and when. Make sure that all affected parties in your organization see a proposed merge. For example, perhaps a network change should be approved not only by the system administration team but also by the networking team and the security team. A rule can take the form of a "minimum number of approvals," but that doesn't limit the number of approvals to the minimum. And while you'll have multiple approvers, you should allow only a handful of people to actually merge the change.

Summary

In this chapter, we explored how Git workflows differ from traditional version control uses. I went over separating your application code from its GitOps configuration manifests. I also touched on how you shouldn't use branches for different environments, which might seem counterintuitive to many but is essential to GitOps. I also discussed trunk-based development and what to look for when setting up your Git policies and PR merging workflows.

The next chapter takes the ideas laid out here and shows how to apply them to your repositories. Specifically, I will go over the layout of your GitOps directories and some things to keep in mind when creating your GitOps directory structure in your Git repository.

References

[4.1] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[4.2] https://git-scm.com/docs/git-cherry-pick

[4.3] https://git-scm.com/docs/git-diff

[4.4] https://kustomize.io

[4.5] https://helm.sh

[4.6] https://trunkbaseddevelopment.com/

[4.7] https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches#about-branch-protection-rules

Chapter 5

Repository and Directory Structures

When adopting GitOps, organizations must plan carefully to divide tasks and configuration files appropriately between repositories and directories in each repository. Standard practices have long existed for using Git-based workflows for infrastructure and software delivery. But with the dawn of cloud-native architectures and Kubernetes, you can now automate a wide range of deployments based on declarations stored in a Git repository.

The question of best practices comes up a lot when creating repositories for GitOps. There is no magic bullet, but several common patterns exist to match the various ways the organization interacts internally.

The overarching consideration when choosing a GitOps directory structure seems to fall under Conway's Law, which states:

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*
>
> *— Melvin E. Conway*

Applying Conway's law to GitOps, we can expect each team of developers to create its own branches and directories within a repository. Furthermore, the structure will be dictated by organizational boundaries (which can also be called "points of demarcation"), such as security, operations, regulatory concerns, etc.

In this chapter, we will discuss some best practices when structuring Git repositories and how you might arrange them based on experiences from different types of organizations.

One final note before diving in: These examples are designed to be a starting point, not a reflection of how your final repository will be represented.

Best Practices

The following are general best practices when it comes to structuring your Git repositories for GitOps. They are designed to be generic to all GitOps implementations and are not tied to a particular toolset or technology.

DRY

The acronym "DRY" stands for "Don't Repeat Yourself." We can adapt it to a GitOps model by rewording it as "Don't Repeat YAML." The idea is simple here; as described in the Templating chapter, storing everything in Git can sometimes lead to copying the same YAML over and over again in different places. Use the strategies described in that chapter to avoid duplication of YAML. Specifically, use Kustomize to keep the base configuration of your deployment and then store the deltas as patched overlays.

Parameterize Where You Need To

There are certain situations where patching isn't the best solution. Patching existing YAML is great when you already know the configurations and deltas beforehand. An example of this is the Ingress Object in Kubernetes [5.1]. This configuration has a host field in the YAML manifest that is supposed to be filled in with the fully qualified domain name (FQDN) of the application being deployed. When you are deploying onto many clusters, the FQDNs of each one may not be known beforehand. Parameterizing your configurations makes sense in this scenario. This is where Helm shines, specifically when you use the lookup feature [5.2].

In the end, you will use a combination of tools to get your desired results, as I explained in the Templating chapter. Keep in mind that there is no "absolutely right" method to do things, and a lot will depend on your environment and communication structure. The main point of this is not to copy the same YAML everywhere.

Repository Considerations

Before we go deeper into how your directory can be structured, there's another important consideration to keep in mind: How many repositories are you going to have?

As mentioned previously, this really all depends on how your enterprise is structured and where the boundaries lie. Also, the GitOps tool being used might have some limitations on handling repository structures that require other considerations that are out of scope for this discussion. But taking a high-level look at things, two patterns arise when considering the structure of repositories: monorepo and polyrepo.

Monorepo

In a monorepo environment, all the manifests for the entire environment, including end-user applications, cluster configuration, and cluster bootstrapping, are stored in a single Git repository. This pattern applies not just to one cluster: every potential cluster in your environment is represented in this single repository. Yes, `dev` and `production` would live in the same repo. Figure 5-1 shows the monorepo solution.

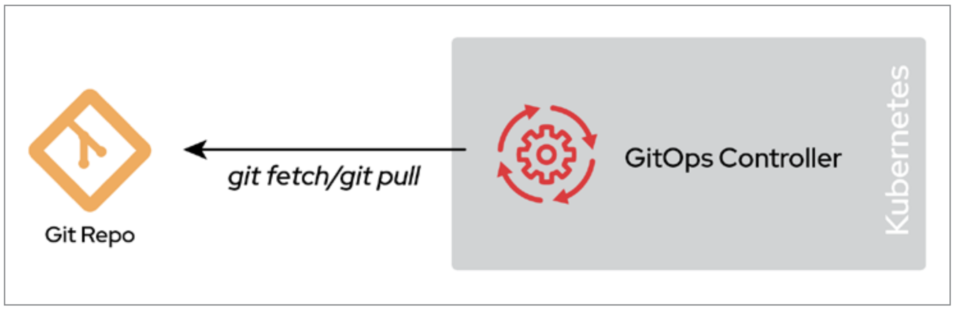


Figure 5-1: GitOps with a monorepo.

The clear advantage of a monorepo is that it provides a central location for configuration changes. This simplicity enables straightforward Git workflows that will be centrally visible to the entire organization, making for a smoother and clearer approval process and merging.

There are several disadvantages, however. The first is scalability. As your organization grows, your environment also needs to grow with it, increasing the overall complexity of each deployment. This can make a monorepo difficult (even impossible) to manage.

There are also performance issues, especially if you use Argo CD [5.3]. As the monorepo grows and changes become more and more frequent, the GitOps controller (for example, Argo CD) takes considerably more time to fetch the changes from the Git repository. This can slow down the reconciliation process and might slow down the correction of deviations from your desired state.

In short, although a monorepo is a valid choice, it can be quickly outgrown by the evolving requirements of the organization's operational needs. It can work if the team managing the environment is small enough and the repository manages only a handful of applications, environments, and clusters. Usually, startups and organizations just starting out with GitOps prefer this approach, which is perfectly valid. Another possible use case is when operating in a lab or another environment with a very limited domain for action.

Polyrepo

A polyrepo environment contains multiple repositories, possibly to support many clusters or deployment environments. The basic idea is that a single cluster can have multiple repositories configured as a source of truth. Figure 5-2 illustrates how multiple repositories can manage a single cluster.

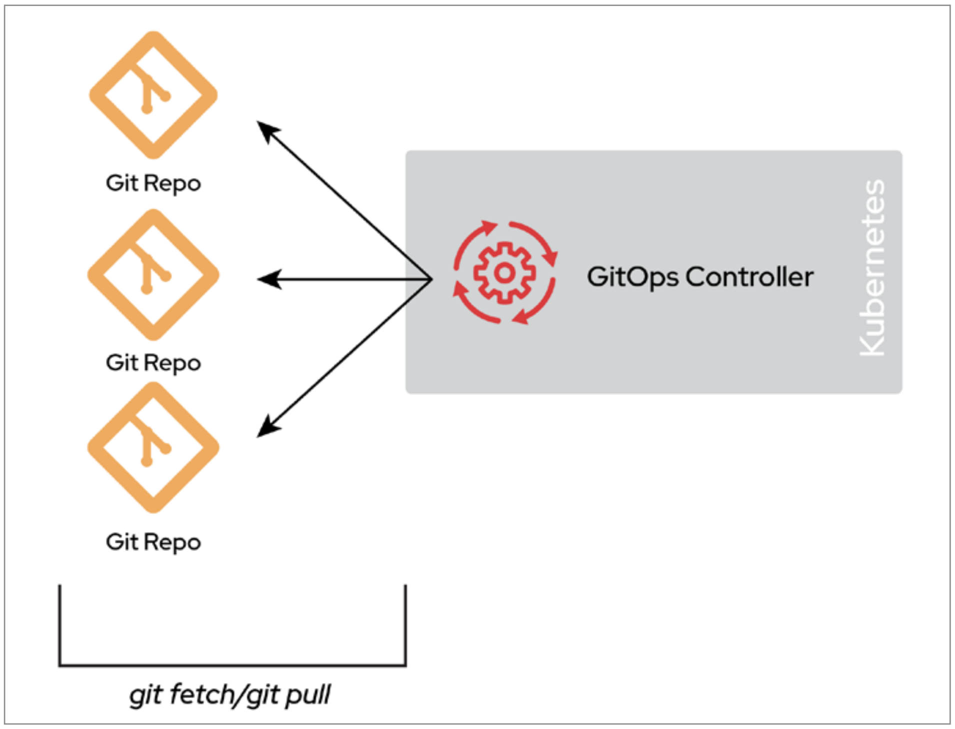


Figure 5-2: The polyrepo environment.

The differences between these Git repositories depend on several factors. A common example is separating concerns between different departments of an organization: a repository for the security team, a repository for the operations team, and one or more repositories for application teams. Another example involves multitenancy, where you have one repository per application.

You could run multiple GitOps controllers within a single cluster, or a GitOps controller can operate in a hub-and-spoke model, as shown in Figure 5-3.

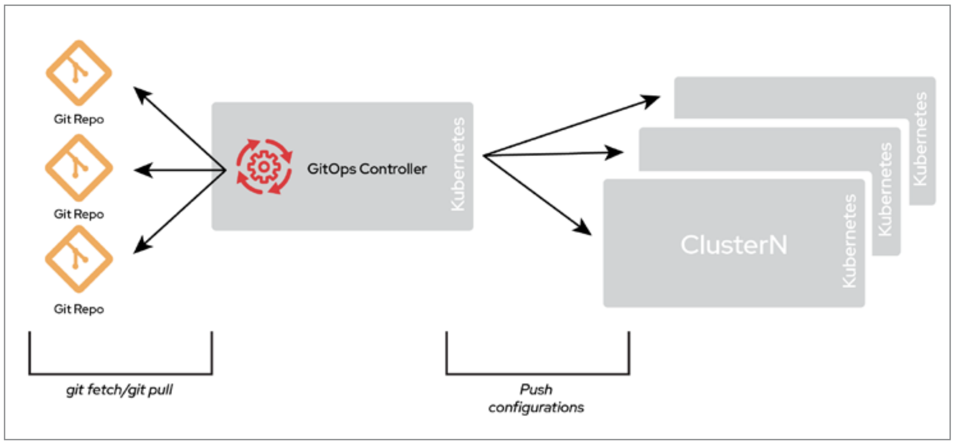


Figure 5-3: The hub-and-spoke model.

A polyrepo, therefore, permits many possible designs.

The primary characteristic of a polyrepo is that not everything is contained within a single repository and that you'll have a sort of catalog of what needs to go into an environment or cluster. The contents of these repositories are the topic of the next section.

One common polyrepo design is *many-to-many*, meaning that each repository points to a single cluster. This is a typical structure in a siloed organization where each team takes care of deploying its own infrastructure.

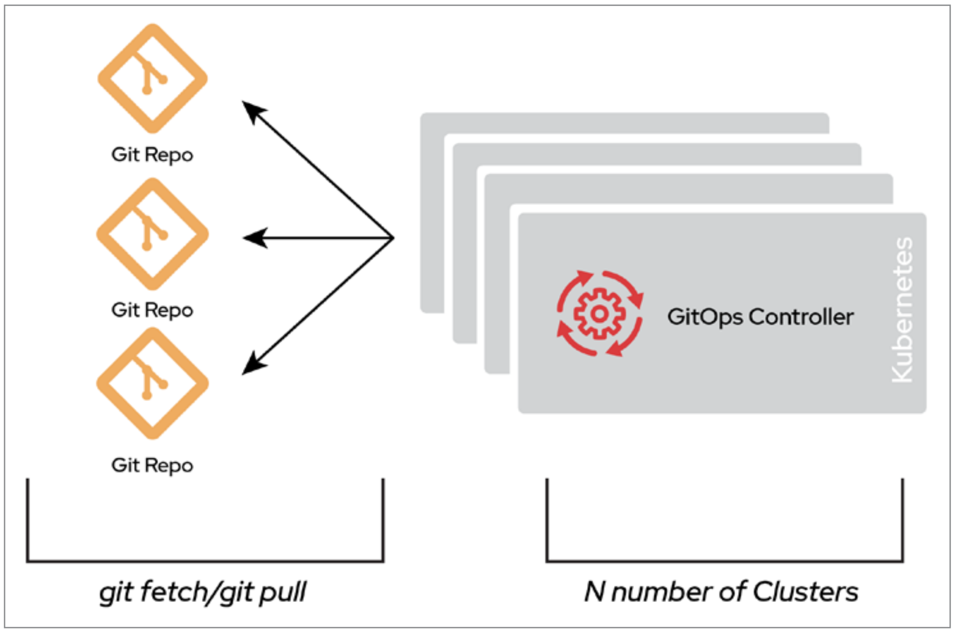


Figure 5-4: The many-to-many model.

The drawback of a polyrepo is that it creates a large number of Git repositories to manage. The number of Git repositories depends on how your organization is laid out and how changes are managed. It's not unheard of for each repository to have its own associated Git workflow. This method can become hard to manage, but it scales incredibly well and is flexible enough to fit almost any organization.

Directory Structures

As explained in the previous section, your Git repository structure will depend heavily on how your organization is laid out. The repositories reflect how your organization communicates with each other and how your current deployment workflow is represented. The different organizations with different workflows are often referred to as *silos*, but more accurately, they are boundaries. For example, developers won't modify platform configurations, whereas operators who work on platform configurations won't go in to change developers' code.

Within each repository, there are many different ways to organize directories. In this section, we'll focus on two use cases, showing both a monorepo and a polyrepo implementation. The polyrepo example shows how organizational boundaries influence the repositories. The monorepo example shows what a repository might look like for a specific cluster.

Repositories Reflecting an Organizational Boundary

This example presents a simple use case with a single organizational boundary between the Kubernetes platform administrator and the Kubernetes application developer. This division between administrators and developers is standard.

Kubernetes Platform Administrator

The repository for the Kubernetes administrator is typically focused on getting the Kubernetes cluster bootstrapped (installed) and configured with the necessary components to run applications. The following is a sample of what the directory structure might look like:

```
├── bootstrap
│   ├── base
│   └── overlays
│       └── default
├── cluster-config
│   ├── gitops-controller
│   ├── identity-provider
│   └── image-scanner
└── components
    ├── applicationsets
    ├── applications
    └── argocdproj
```

Note: The name of the directories are not important; you can change them to suit your needs/preferences. What's important is the layout and what the directories represent. The resources listed at the end of this chapter explain how to use other controllers, such as Flux, instead of Argo CD.

Here is a short explanation of the directories and files in this repository:

- `bootstrap`: This stores bootstrapping configurations. These are items that get the cluster configured with the GitOps controller. The `base` directory contains YAML installation configuration, while `overlays` contain GitOps controller configurations. There is only one overlay, here called `default`, because this is the only overlay in our simple example.

The `default` directory contains a `kustomization.yaml` file that has `components/applicationsets/` and `components/argocdproj/` as a part of its `bases` configuration. It will look something like the following:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../../base
- ../../../components/applicationsets
- ../../../components/argocdproj
```

- `cluster-config`: This is where YAML for the cluster's configuration manifestlives. The manifest determines the behavior of the cluster.

  The files under `gitops-controller` use Argo CD to manage themselves. The `kustomization.yaml` file refers to `bootstrap/overlays/default` in its `bases` configuration. This directory gets deployed as an ApplicationSet in `components/applicationsets/cluster-config-appset.yaml`.
- `components`: This configures the GitOps controller (in this case, Argo CD). `applicationsets` contains YAML for the ApplicationSets and `argocdproj` contains YAML for the Argo AppProject.

The Application configured in the `components` directory can point to other Git repositories. The administrator uses the directory as a "point of entry" to onboard applications.

An administrator bootstraps a cluster by running:

```
$ kubectl apply -k bootstrap/overlays/default
```

This command loads in all the configurations and deploys the cluster-specific configurations onto the Kubernetes cluster.

Kubernetes Application Developer

The repository for the Kubernetes application developer is actually pretty straightforward. It's one of the last components that lands on a cluster, so it lets you be very terse and make assumptions about what previous configurations have already done. Most of the groundwork has already been implemented on the clusters by other personas. A typical directory structure is:

```
└── deploy
    ├── base
    └── overlays
        ├── dev
        ├── prod
        └── stage
```

The bulk of the configuration is in the `base` directory, and only the deltas are stored in `overlays`. This example shows environments such as `dev` and `prod`, but can also be other configurations (such as `clusters`). This layout does make a lot of assumptions, but the idea is that the cluster will already come configured.

The YAML files stored in this layout won't state a namespace in their `metadata` sections. This is because the creation and management of namespaces are typically controlled by the Kubernetes administrator. This may change from organization to organization, so it is not a hard and fast rule, but it's something to keep in mind and communicate about.

Although the GitOps tool deploys the application, an example deployment into the **dev** environment would look like this:

```
$ kubectl apply -k deploy/overlays/dev
```

The `dev` overlay consumes all the YAML in `base` and overlay the deltas. Since most GitOps tools support Kustomize, the combination allows a flexible deployment. For example, if one cluster is administered through Flux and another through Argo CD, this structure would work for both clusters.

Other Boundaries

The previous section showed a simple use case to illustrate how the division of responsibilities can dictate the contents and structure of the repositories. Best practices are similar when an organization has more than just two boundaries. Other common roles that define boundaries are Kubernetes service SRE, Kubernetes security team, and application release manager.

GitOps Repo Example

This example shows how a repository can be laid out using the DRY principle and keep the structure generic enough to deploy to many clusters. This example also assumes “full DevOps,” where the entire organization (both Kubernetes administrators and Kubernetes developers) is taking part, working together in the release process.

This example, like the previous one, is based on using Argo CD as the GitOps controller:

```
├── bootstrap
│   ├── base
│   └── overlays
│       └── default
├── components
│   ├── applicationsets
│   └── argocdproj
├── core
│   ├── gitops-controller
│   └── sample-admin-workload
└── apps
      ├── bgd-blue
      │   ├── base
      │   └── overlays
      │       ├── dev
      │       ├── prod
      │       └── stage
      └── myapp
          ├── base
            └── overlays
          ├── dev
          ├── prod
          └── stage
```

The basic components of the structure are:

- `bootstrap`: This plays the same role as the `bootstrap` directory in the previous example.
- `components`: This plays the same role as the `components` directory in the previous example. Manifests that can live here include role-based access control (RBAC), Git

repository secrets, and configuration files specific to the Git controller, Argo CD. Each configuration has its own directory.

- **core**: This contains YAML for the core functionality of the cluster. The Kubernetes administrator places resources here that are necessary for the functionality of the cluster, such as cluster configurations and cluster workloads.

  The files under **gitops-controller** use Argo CD to manage themselves. The **kustomization.yaml** file refers to **bootstrap/overlays/default** in its **bases** configuration. This directory gets deployed as an ApplicationSet in **components/applicationsets/core-components-appset.yaml**.
  To add a new "core functionality" workload, the administrator adds a directory with YAML content in the **core** directory.

- **apps**: This is where the workloads for this cluster live. Similar to **core**, this directory gets loaded as part of an ApplicationSet under **components/applicationsets/tenants-appset.yaml**.

  The **apps** directory is where developers and release engineers work. They just need to commit a directory with some YAML, and the ApplicationSet takes care of creating the workload.

  The **bgd-blue/kustomization.yaml** file can point to another Git repository. Thus Kustomize helps you your YAML in many reposistories, if this is convenient. The **bgd-blue** directory can also be a Git submodule.

Conclusion

In this chapter, we discussed best practices for creating GitOps repository and directory structures. Although there are generic examples that you can follow, there is no one answer. Your directory structure is going to be driven by your organizational structure and possibly regulatory considerations as well. However, following these basic best practices can help lead you in the right direction.

To get you started working with GitOps directory structures, I provide several starting points in the following repositories:

- Using Argo CD on Kubernetes [5.4]
- Using Flux on Kubernetes [5.5]
- Using OpenShift GitOps on OpenShift [5.6]

References

[5.1] https://kubernetes.io/docs/concepts/services-networking/ingress/

[5.2] https://helm.sh/docs/chart_template_guide/functions_and_pipelines/

[5.3] https://argo-cd.readthedocs.io/en/stable/operator-manual/high_availability/#monorepo-scaling-considerations

[5.4] https://github.com/christianh814/example-kubernetes-go-repo

[5.5] https://github.com/christianh814/example-kubernetes-goflux-repo

[5.6] https://github.com/christianh814/example-openshift-go-repo

Chapter 6

CI/CD with GitOps

CI/CD (continuous integration/continuous delivery) has become the workbench for DevOps practitioners. In Chapter 1, we briefly went over what CI/CD means and where GitOps fits into your CI/CD workflows. In this chapter, we will dive deeper into where GitOps fits in your CI/CD pipelines, explore the various ways you can implement it, and go over the pros and cons of each method. Along with directory structures, CI/CD with GitOps is one of the topics people ask about most when discussing GitOps workflows.

CI and CD Can Be Decoupled

When looking at how GitOps approaches CI/CD, you first have to understand that continuous integration and continuous deployment aren't necessarily tied as closely together as developers traditionally thought.

People typically glom CI and CD together, because popular DevOps tools like Jenkins aim to do both. Jenkins has successfully created a platform to manage both CI and CD using a single management system. But with the advent of Kubernetes, microservices, and cloud-native architecture, many paradigms are starting to be decoupled, including CI/CD.

Integrating Synchronous and Asynchronous Tools

CI is primarily a synchronous process that has a start and an end, and that is usually triggered by some event. For example, a commit to a specific repository or branch can trigger a pipeline that builds the application, runs tests against it, and maybe triggers other pipelines. The key point here is that the process is synchronous, and there are typically dependencies between each step.

GitOps, conversely, is asynchronous. As discussed in Chapter 1, the fourth GitOps principle states that software agents continuously observe the actual system state and attempt to apply the desired state. Simply put, the GitOps controller has no idea what is taking place in the CI system. The GitOps controller only looks at the Git repository (which represents the desired state) and acts only if the desired state has mutated.

Integrating synchronous tools with asynchronous tools can be challenging, especially when you are trying to integrate systems that have been in place for a while. In this chapter, we'll look at three methods for integrating GitOps with your CI/CD system.

CI Managed

In the CI-managed model, the CI tool owns and manages the complete deployment using one system. This is close to what most people are used to; for instance, the popular Jenkins tool mentioned earlier owns the entire process. In this model, the GitOps process is orthogonal to the CI system, and the system is usually managed using floating tags (such as `dev`, `stage`, and `prod`). It's the CI's job to tag the appropriate release. Figure 6-1 shows this mostly linear design.

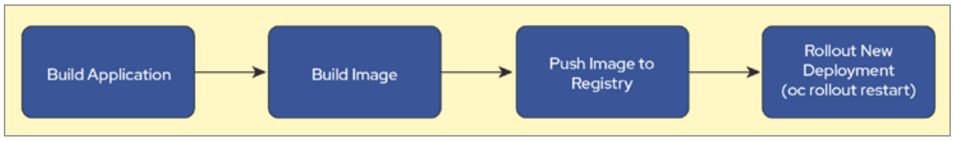


Figure 6-1: The CI-managed CI/CD model.

Benefits

The main benefit of this model is that it follows the traditional CI/CD design that many organizations are already using. Matching an existing model reduces the need to re-architect your processes, and existing systems like Jenkins can stay in place without much modification. Many organizations start with this model when they are early in their exploration of GitOps and its practices. This model is also where people start when they have existing pipelines that they simply want to reuse.

Drawbacks

The main drawback is that the model goes against the spirit of GitOps, philosophically. If the idea is to use Git as the source of truth, CI-managed CI/CD undermines that goal through its dependence on floating tags. With floating tags, someone can forcibly push an update to the tag and change the system's state without the GitOps controller knowing about it because the Git repository simply trusts what the tag says. For example, if someone tags an image as `prod` and pushes it to the registry without passing the image through the proper CI process, the GitOps controller will deploy that image in its `prod` deployments, with all the risks of evading the CI build process.

CI Owned and CD via GitOps

In this model, the CI platform owns the process of building, managing, and deploying new versions of an application but uses the GitOps controller for the actual deployment. Once the GitOps controller deploys the new version of the application, it hands the process back to the CI platform that actually validates and tests the update. In this model, the CI process is more hands-on with the GitOps controller, and integrations will need to be written if plug-ins are not already available.

For example, the CI process builds an image and pushes it to a registry, after which the CI system changes the appropriate tag in Git to trigger deployment from the GitOps controller. The CI system also monitors the deployment and verifies it once it has rolled out. In this way, the whole process can be thought of as synchronous (Figure 6-2).

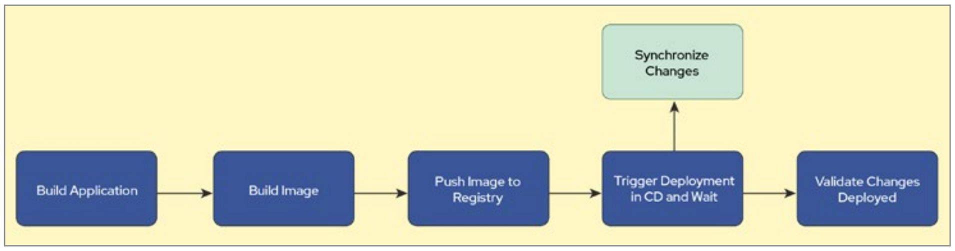


Figure 6-2: The CI-owned and GitOps-deployed model.

Benefits

The advantage of the CI-owned and GitOps-deployed method is that it tries to comply with the GitOps principle that the state stored in Git is the state that ends up being deployed to the cluster. This way, you know exactly what gets deployed and what is running in your cluster. Another advantage is that the method keeps the linear aspect of a pipeline intact. This simplicity makes things easier to follow, and easier to troubleshoot when something goes wrong.

Drawbacks

The drawback to this model is that integrating the pipeline with the GitOps controller adds a layer of complexity. You are mixing the synchronous CI actions with asynchronous Git-based processes. This combination can be challenging and, in many cases, will require refactoring a lot of things in your pipeline.

This design makes the most sense when there is no gating between the environments (like dev ~> test). This is getting closer to fully automating the process.

CI Triggered and GitOps Owned

With the CI triggered and GitOps owned method, the CI platform that builds the application relies on the GitOps controller to handle the deployment in an asynchronous fashion. This is done mainly by having the CI process create a GitHub pull request (PR) against the tracked branch for each stage of the pipeline.

There are other ways to implement this model, too. In "fire-and-forget," the CI tool commits changes to the tracked repository automatically (i.e., without gating by issuing a PR). Once the GitOps controller completes the deployment, a postsync hook can trigger additional pipelines to perform any post-deployment tasks. The model is illustrated in Figure 6-3.

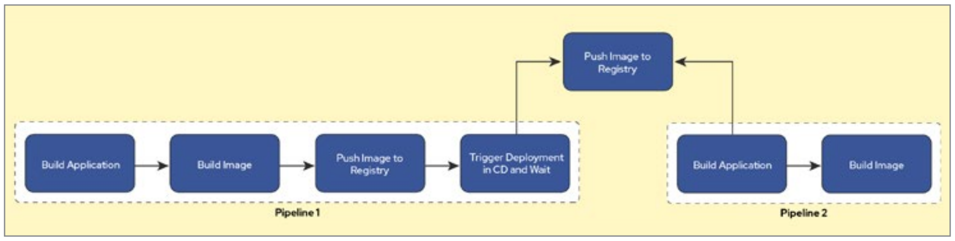


Figure 6-3: The CI-triggered and GitOps-owned method.

Benefits

The major benefit of this design is that the CI process, which is synchronous, is decoupled from the asynchronous GitOps controller. This lets you avoid the challenges you face when trying to integrate the two. The model lets each system do what it does best (aka "the best tool for the job").

Drawbacks

The drawback to this approach is that the pipeline ceases to be a simple linear process. The loosely coupled processes can make it difficult to troubleshoot and keep track of where a build is in its process.

Also, a sync process can be triggered frequently for other reasons (like a self-heal when a resource is missing), so you need to code the pipelines to account for this.

A Mindset Shift

Using GitOps with CI/CD calls for a shift in thinking. Traditionally, you think of promoting code or images into environments. But GitOps with CI/CD changes that mindset. When working with GitOps workflows, it's not code but manifests that get promoted and tested across your environments. A code update triggers an image build, which ends up getting updated in the manifest. Any other change in the manifests should follow the same promotion process as an image update.

So when you're thinking about promoting your application from environment to environment, think about it instead as promoting *manifests*. Because Kubernetes relies on immutable containers, the code isn't what really gets promoted; rather, it's the desired state of the application running on the cluster. This is a subtle change in mindset, but an important one.

Conclusion

In this chapter, we explored a few different ways you could integrate your CI system with the GitOps paradigm of doing CD. We went over how you can plug your GitOps controller into your existing CI/CD system, how to take a synchronous CI system and tie it into your asynchronous GitOps deployment system, and finally, how to run a decoupled CI and CD system using GitOps.

In the next chapter, we will dive into a vital security-related topic that, like CI/CD integration, challenges some basic GitOps precepts: how to manage secrets in a GitOps-friendly way.

Chapter 7

Handling Secrets

Along with Git directory structures and CI/CD, another big question people raise about GitOps is what to do with Kubernetes secrets. The GitOps principles state that a system's state is stored declaratively, and this rule also applies to Secrets.

This requirement poses a challenge for both development teams, which store application configurations, and system operations teams, which store infrastructure configurations. Applications use sensitive data, notably the token or credentials that grant access to a database. For operations, the critical secret could be information about accessing other resources in the environment or other tokens or credentials. Storing such sensitive information in a Git repository—even a private one behind a firewall—poses a security threat. Git secrets shouldn't be stored in clear text, even in a private, hosted repository.

In this chapter, I will go over how to securely handle secrets and comply with the GitOps principles.

Common Patterns

Storing secrets securely in a version control system isn't a new problem; it existed with all of the Infrastructure as Code tools in the past and was later inherited by Kubernetes. Two common methods have emerged to use Kubernetes secrets securely in GitOps workflows:

- Storing encrypted secrets in Git.
- Storing secrets in external services and storing references to the secrets in Git.

The rest of this chapter looks at each of these patterns in detail.

Storing Encrypted Secrets

Storing encrypted secrets is one of the most popular ways to avoid exposing sensitive data. This solution involves encrypting the secret before uploading it into Git. Later, when the secret is applied, the user process decrypts it. The secret is never stored anywhere in plaintext.

There are a few popular technologies for encrypting secrets. In this chapter, we will focus on the most popular one: Sealed Secrets for Kubernetes [7.1] by Binami.

Sealed Secrets

Sealed Secrets is a system that runs in your Kubernetes cluster to encrypt and decrypt secrets using asymmetric cryptography (also known as *public-key cryptography*). Sealed Secrets has become very popular, especially among GitOps practitioners, because of its ease of use and low barrier to entry. Thanks to its not-so-steep learning curve, it has been widely adopted by Kubernetes administrators.

Sealed Secrets has two primary components. The first is a controller that runs inside of Kubernetes to manage the public and private keys. This controller is also responsible for decrypting the encrypted secrets. The second component is a command-line interface (CLI) client called `kubeseal`, used by end users to encrypt secrets.

The Sealed Secrets controller watches for the existence of a `SealedSecret` resource when it gets applied to the Kubernetes cluster. The controller then decrypts the data and loads the corresponding secret into the Kubernetes cluster. The process is illustrated in Figure 7-1.

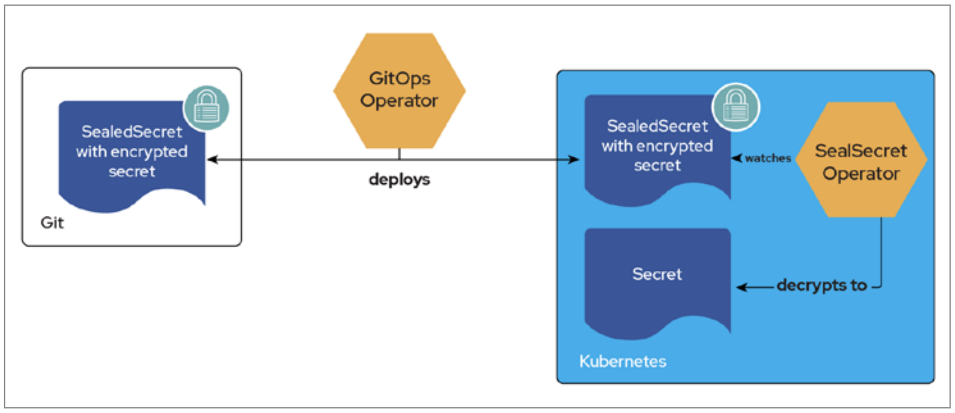


Figure 7-1: The Sealed Secrets controller in action.

The controller uses a random cryptographic nonce when encryption is done, further hardening the system.

`kubeseal` is the CLI tool used by an end user to interact with the Sealed Secret controller. The user runs `kubeseal` to encrypt a secret before it is added to the Git repository. The nice thing about this system is that the end user doesn't have to possess or know the public key because the CLI tool connects to the Sealed Secret controller to do the encryption. (Figure 7-2).

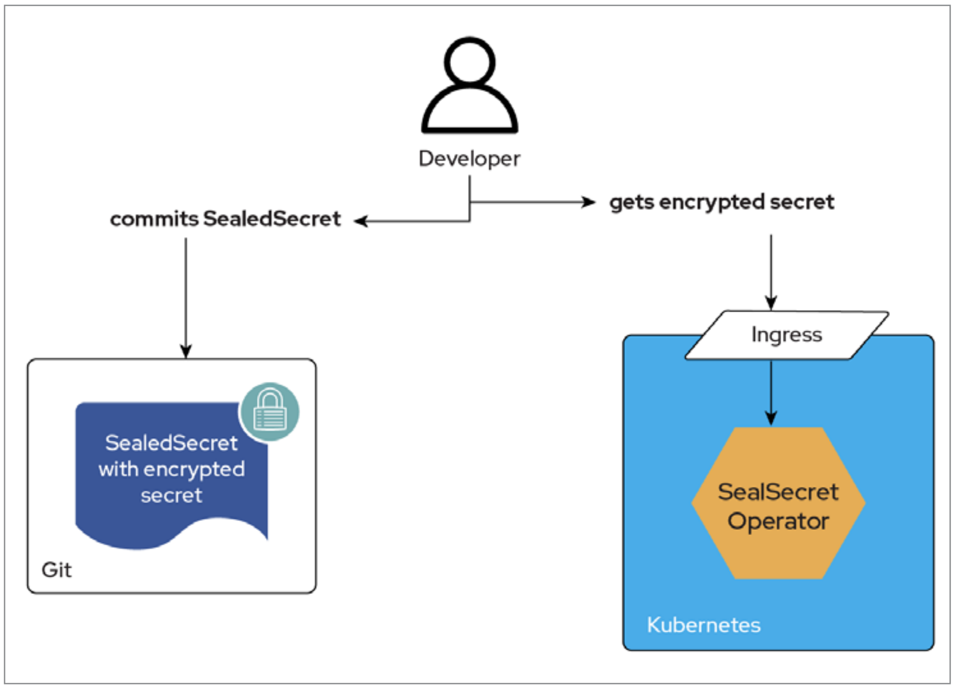


Figure 7-2: The kubeseal workflow.

You can also provide a public key in the CLI for "offline" encryption, where you don't use the Kubernetes cluster. This alternative is useful for automation.

Sealed Secrets supports automatic rotation of the encryption keys and, optionally, deprecation of past keys.

Challenges of Storing Encrypted Secrets

Storing encrypted secrets in Git presents a few challenges. The first is that someone or something needs to generate the encrypted secret beforehand. Because the secret has to exist before encrypting it, there is a danger that whoever generates it will mistakenly commit the plaintext secret into Git.

There is also the private key or "key 0" problem, which refers to the private key that is stored on the system. If you're managing many clusters, managing the storage process at scale can become an issue. This private key needs to be installed on every Sealed Secret controller. Furthermore, if that key is rotated or expired, you need to make sure that every cluster gets the new key in order to work. The number of updates required for each change can lead to a lot of overhead in managing the Sealed Secret controllers in your environment. Also, the private key is stored in `etcd`, which creates another attack vector.

Still, storing encrypted secrets is a valid way to solve security problems, and I've seen it used successfully in many environments.

Storing Secret References

In this methodology, secrets are not stored in Git (neither in plaintext nor in an encrypted format) but rather in a secret management system. Only a reference to the secret is stored in Git. You can safely store this reference because the reference contains no sensitive information. When the secret is needed to perform operations, a controller "fetches" the actual secret and applies it to the cluster.

Many tools provide this function, and in the next section, I will focus on one that hopes to broker all solutions: External Secrets [7.2].

External Secrets

External Secrets is a Kubernetes controller that integrates external secret management systems such as AWS Secrets Manager [7.3], HashiCorp Vault [7.4], Google Secrets Manager [7.5], and Azure Key Vault [7.6] using a plug-in model. The External Secrets controller reads information from external APIs (for example, HashiCorp Vault) and injects the values from the external system into Kubernetes as a secret.

This process works by providing the controller with an `ExternalSecret` object. This object has a `secretStoreRef` field where a user defines a `SecretStore`, which has information about how to fetch the required secret. `ExternalSecrets` resources, as I mentioned before, can be stored in Git, as they don't have any sensitive information. `ExternalSecrets` itself does not perform any cryptographic operations and instead fully relies on the backends. The general process is shown in Figure 7-3.

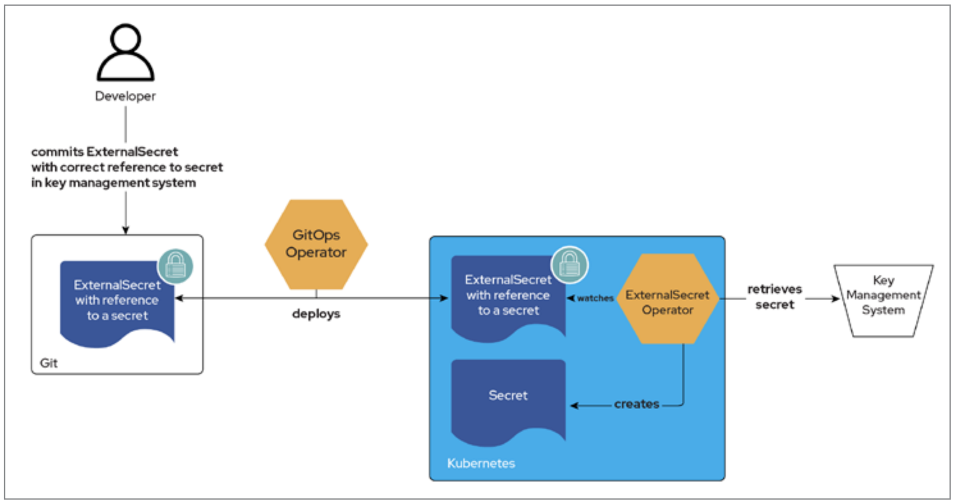

Developer
commits ExternalSecret with correct reference to secret in key management system
ExternalSecret with reference to a secret
Git
GitOps Operator
deploys
ExternalSecret with reference to a secret
watches
ExternalSecret Operator
retrieves secret
Key Management System
Secret
creates
Kubernetes

Figure 7-3: External Secrets operation.

The external secrets controller is also multitenant, so secrets cannot be read across namespaces. There's even a namespaced scoped mode where you can have an External Secrets controller in each namespace that needs it.

External Secrets is popular because it abstracts the backend secret management system so users can use the same essential procedure independent of the secret provider. Furthermore, External Secrets is useful for sites with many different secret sources.

Challenges of Storing Secret References

The first issue that comes to mind with respect to secret references is that the secret is actually managed outside the scope of GitOps. This can be problematic because the second GitOps principle states that the resource should be versioned. Since you are only referencing the secret in Git, you are not really keeping track of what version of that secret you are using. Another challenge is that the management of the secret is done outside of the Kubernetes platform itself. This separation might require another team to manage the life cycle of that secret.

That being said, storing secret references is a good solution, especially if you don't want to get the end user involved in managing the life cycle of the secret.

Conclusion

In this chapter, we examined two different methods of managing secrets using GitOps. The first approach is to store an encrypted version of the secret in your Git repository. The second method is storing a reference of the secret in Git while another platform manages that secret. Both solutions are quite popular, and what works for you depends on the security practices in your specific environment.

References

[7.1] https://github.com/bitnami-labs/sealed-secrets

[7.2] https://github.com/external-secrets/external-secrets

[7.3] https://aws.amazon.com/secrets-manager/

[7.4] https://www.vaultproject.io/

[7.5] https://cloud.google.com/secret-manager

[7.6] https://azure.microsoft.com/en-us/services/key-vault/

Chapter 8

Other Considerations

In this book, I have taken you on a step-by-step journey toward building GitOps pipelines, focusing on environments running containers on a Kubernetes cloud-native infrastructure. These are the environments that produced GitOps (see Chapter 1, *What Is GitOps?*). And this history is the main reason why, when you search for GitOps, you will find a lot of information about Kubernetes, Kubernetes-native tools, and how to integrate them with your GitOps practices.

However, other considerations exist when adopting GitOps and integrating it with your current environment. We'll cover them in this chapter. Some of these considerations reach into other aspects of your infrastructure that might not be GitOps-ready. Luckily, many of these GitOps paradigms have been around a while, so you'll find a lot of integration points that fit naturally with your environment. This chapter introduces open source tools and Red Hat services to help achieve advanced goals.

Multicluster Management

GitOps has its foundations in Kubernetes, and increasing Kubernetes adoption across the industry will lead more and more sites to run multiple Kubernetes clusters. These clusters could be in a single datacenter on-premises, across multiple regions on a hyperscale, or in a hybrid cloud that combines the two types of locations. Reasons for running multiple clusters include multitenancy, disaster recovery, and isolation for the sake of security. Regardless of the reason, we now need to manage the life cycle of these Kubernetes clusters from creation to retirement.

Red Hat Advanced Cluster Management for Kubernetes [8.1], based on the open source CNCF sandbox project Open Cluster Management [8.2], was built for this exact use case. With Red Hat Advanced Cluster Management for Kubernetes, users can install OpenShift clusters, manage policies, and manage the entire life cycle of their clusters from a single pane of glass. You can manage these clusters via declarations read from a Git repository, and the platform has native integration with Argo CD for Application deployments across all clusters. Furthermore, you can add and manage other *KS clusters from various hyperscalers: Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Google Kubernetes Engine (GKE), etc.).

By combining Red Hat Advanced Cluster Management for Kubernetes with your GitOps workflows, you can manage the complete application life cycle, including the underlying infrastructure.

Non-Declarative Infrastructure

Even with the mass adoption of containers, we will still be living in a world of virtual machines for the foreseeable future, and even more so with bare metal servers. Therefore, although Kubernetes and cloud-native infrastructure strongly emphasize declarative configuration, you will still find yourself integrating and managing many components that are non-declarative.

Red Hat Ansible Automation Platform [8.3] provides a framework for building out your enterprise automation at scale. With Red Hat Ansible Automation Platform, users can integrate existing Ansible automation playbooks into their GitOps work-flows in various ways, using tools such as the Ansible Operator SDK, Ansible Tower, and Ansible Runner integration points. I think it makes the most sense to integrate components using the native integration with Red Hat Advanced Cluster Management for Kubernetes [8.4]. The built-in native integration gives you a central way to manage your cloud-native infrastructure alongside your traditional, non-declarative infrastructure.

By using Red Hat Ansible Automation Platform in your GitOps workflows, you pull your traditional infrastructure and applications into the purview of your GitOps workflows. This lets you take advantage of the power of Git-based workflows on your Kubernetes cloud-native infrastructure.

Security

Security is a broad and complicated topic to discuss holistically. Within the context of GitOps, the conversation has more to do with container and Kubernetes security practices. Red Hat Advanced Cluster Security for Kubernetes, based on the open source project StackRox [8.5], is another great integration point with your GitOps workflows. Red Hat Advanced Cluster Security for Kubernetes [8.6] provides toolsets to address the security needs of running infrastructure on Kubernetes.

Red Hat Advanced Cluster Security for Kubernetes offers visibility into the security of your cluster, vulnerability management, and security compliance through auditing, network segmentation awareness and configuration, security risk profiling, security-related configuration management, threat detection, and incident response. The service can be configured to identify vulnerabilities in your containerized workloads so you have visibility into what is going into your clusters.

Red Hat Advanced Cluster Security for Kubernetes has many integration points, but I see it as living in the CI/CD process. Scanning images before they are deployed can help mitigate security breaches before they happen. Also, continuously scanning images in your environment alerts you to vulnerabilities faster so you can take action sooner.

Integrating Red Hat Advanced Cluster Security for Kubernetes into your CI/CD process brings in security earlier in the process. This methodology is known as DevSecOps [8.7].

Base Image Selection

Base image selection is an important factor in security. It’s a lot easier to manage updates to applications when they are all built using the same base image. Unifying the containers on one base image lets your organization build a “golden image” undergirding all of its applications. The image not only has the needed toolsets for every application but includes all required security patches. When you update your base image, every application built from that moment on will have all the necessary security patches. You can take advantage of this efficient security practice by using Red Hat Universal Base Image [8.8].

Whether or not you’re a Red Hat customer, Red Hat Universal Base Image (UBI) gives you greater reliability, security, and performance afforded by official Red Hat container images running OCI-compliant Linux containers. You can take the same bits that are used to build Red Hat Enterprise Linux and use them in your containerized applications. You can build a containerized application on UBI, push it to your chosen container registry server, and share it. The Red Hat Universal Base Image allows you to build, share, and collaborate on your containerized application wherever you want at no cost, including OpenShift Kubernetes, *KS Kubernetes, and Docker Compose environments.

Using a secure base image and building policies that ensure the use of that base image (and keeping the base image updated) helps mitigate the risk of stray applications built without security patches. Even if one does slip through the cracks, you can put policies in place to flag those images using Red Hat Advanced Cluster Security for Kubernetes.

Everything as Code

Taking a step back, how do all these toolsets fit in your environment? GitOps has evolved every aspect of "as-code" paradigms to fully embrace automation. With these new considerations, as these technologies progress, you can now do "everything-as-code." In this context, the term means that your entire CI/CD process is described declaratively in a Git repository (Figure 8-1).

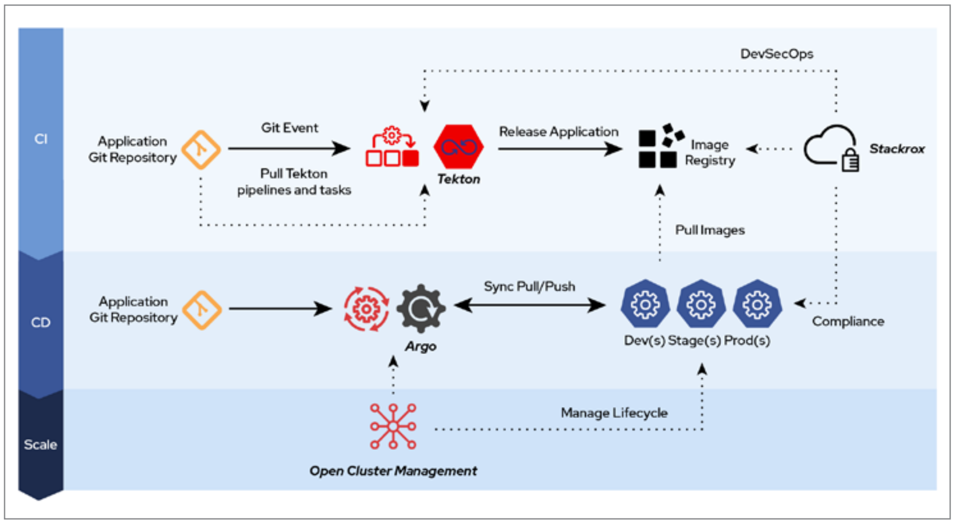


Figure 8-1: Everything as code.

In Figure 8-1, your CI process is handled by the cloud-native CI/CD platform Tekton, where you can set up Pipeline-as-code [8.9] to host your pipelines in Git (similar to a GitHub action) and have Tekton read your Pipeline definitions from Git.

The StackRox platform manages your security by scanning your image repository. It can interact with Tekton to fail promotion if there's a vulnerability found. StackRox also interacts with your Kubernetes cluster to continuously monitor your deployments as they run.

Argo CD can manage your clusters in the CD layer by reading the application deployment configuration from a Git repository (this should start to sound familiar, now that you understand how GitOps works). It keeps everything in sync as it works with the Open Cluster Management server to create and manage your clusters at scale using these Git repositories.

In short, you can set everything up in a declarative manner using cloud-native toolsets. This "Everything-as-Code" configuration is the holy grail of DevOps, representing what a lot of us DevOps practitioners have been striving for: full, declarative automation for the entire organization.

Conclusion

This chapter covered ancillary considerations to think about when integrating a GitOps workflow into your organization. We discussed how Red Hat Advanced Cluster Management for Kubernetes helps you create, manage, and scale Kubernetes clusters from a single control plane. We also saw how you can incorporate security practices into your CI/CD workflows with Red Hat Advanced Cluster Security for Kubernetes.

You can also use the Red Hat Ansible Automation Platform to integrate things that lie outside your GitOps workflows. Finally, we took a look at how all these technologies can be used together to provide an "Everything-as-Code" design, where you can declaratively manage your environments.

References

[8.1] https://www.redhat.com/en/technologies/management/advanced-cluster-management

[8.2] https://open-cluster-management.io/

[8.3] https://developers.redhat.com/products/ansible/overview

[8.4] https://www.redhat.com/en/about/videos/acm-ansible-integration-overview

[8.5] https://www.stackrox.io

[8.6] https://www.redhat.com/en/resources/advanced-cluster-security-for-kubernetes-datasheet

[8.7] https://developers.redhat.com/topics/devsecops/

[8.8] https://developers.redhat.com/blog/2021/04/13/how-to-pick-the-right-container-base-image

[8.9] https://cloud.redhat.com/blog/create-developer-joy-with-new-pipelines-as-code-feature-on-openshift

About the Author

Christian Hernandez leads developer experience at Codefresh. He previously spent eight years on the customer and field engagement team in Red Hat's Hybrid Platforms organization and hosted the biweekly GitOps Guide to the Galaxy [1]. A contributor to Argo CD and OpenGitOps, Christian is a technologist with experience in infrastructure engineering, systems administration, enterprise architecture, tech support, advocacy, and management. Lately, he has been focusing on Kubernetes, DevOps, cloud-native architecture, and GitOps practices. Christian is passionate about open source and containerizing the world one application at a time.

Reference

[1] https://www.youtube.com/playlist?list=PLaR6Rq6Z4IqfGCkI28cUMbNhPhsnj4nq3