Florian Beetz · Anja Kammer · Simon Harrer

GitOps

Cloud-native Continuous Deployment

Quickstart with Kubernetes

INNOQ

GitOps

Cloud-native Continuous Deployment

Florian Beetz
Anja Kammer
Dr. Simon Harrer

ISBN 978-3-9821126-8-8
innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · WWW.INNOQ.COM

Layout: Tammo van Lessen with XeLaTeX
Design: Sonja Scheungrab
Print: Pinsker Druck und Medien GmbH, Mainburg, Germany

GitOps – Cloud-native Continuous Deployment
Published by innoQ Deutschland GmbH
1st Edition · 2021-07-15

Contents

Acknowledgments

This book has a bit of a history. Back in 2019, Simon hosted a seminar for master students in computer science on all things Kubernetes at the University of Bamberg, Germany. Florian participated in the seminar and choose the topic of evaluating whether GitOps is an evolution of DevOps. After Florian handed in his great seminar thesis and gave his talk on his topic within the seminar, it became clear Simon and Florian wanted to share what they learned so far. That's when they began to write up everything for their website gitops.tech[1]. At first, Anja just wanted to do a review but then joined our effort in 2020, and we began our journey towards this book using the material from the website as a starting point. But without that seminar in 2019, without that Florian choosing GitOps as the topic of his seminar thesis, this book would not exist today. So thank you, Prof. Dr. Guido Wirtz for allowing this seminar to happen!

We'd also like to thank the many people that gave honest feedback, critic, and even praise for early versions of this book (in alphabetical order): Linus Dietz, Vincenzo Ferme, Dr. Michael Oberparleiter, Alexis Richardson, Gregor Riegler, Hans-Christian Sperker, Dr. Andreas Schönberger, Tammo van Lessen, and Daniel Westheide.

And last but not least, we'd like to thank Sonja Scheungrab and Susanne Kayser for the great cover work.

[1] https://gitops.tech

Welcome

This book is our management summary of GitOps. It's our understanding of what GitOps is, why we like it so much, and how it works. It covers a comparison of GitOps to various other terms like DevOps or NoOps and aims to answer the typical questions when being first confronted with GitOps and one asks when diving deeper into the world of cloud-native continuous deployment. If you want to get yours hands dirty straightaway, there is a tutorial waiting for you in the back pages.

Our main goal is to get you up to speed in discussions about GitOps, and if you follow the tutorial you'll also get a feel for it. That's why this book is really for a wide range of people who build software today: from software engineer to team manager, from product owner to site reliability engineer. We think it helps to evaluate whether GitOps might help you in your current project (or not) and how you would start applying it in your setting if you think it can move you forward.

The book, however, is not an absolute truth, no bible of any sort. Don't follow it blindly hoping for some miracle. As any advice, take it with a grain of salt as it might not be the solution for you right now. With this in mind, happy reading!

The field of GitOps is moving fast. In such a fast moving field like GitOps, a writing such as a book like ours can become outdated quickly. That's why we need your help. If you've found something that's out of date, think that we should include some new tools or concepts, or just found a typo, please tell us by creating an issue in the accompanying GitHub repository[a]. It's, however, not just about keeping the content relevant, but to build up a community to discuss ideas around GitOps and perhaps find even better ways to express what GitOps really is all about. So feel free to start a discussion[b] anytime.

[a] https://github.com/gitops-tech/book

[b] https://github.com/gitops-tech/book/discussions

What is GitOps?

GitOps is a way of implementing Continuous Deployment for cloud native applications, coined by Weaveworks in 2017[2]. It focuses on a developer-centric experience when operating infrastructure, by using tools developers are already familiar with, such as Git, Infrastructure as Code, and Continuous Integration.

The core idea of GitOps is having a Git repository that contains declarative descriptions of the infrastructure currently desired in the target environment and an automated process to make the environment match the described state in the repository. If you want to deploy a new application or update an existing one, you only need to update the repository — an automated process handles everything else. It's like having cruise control for managing your applications.

GitOps: versioned CI/CD on top of declarative infrastructure. Stop scripting and start shipping.

Kelsey Hightower[3]

[2] https://www.weave.works/technologies/gitops/

[3] https://twitter.com/kelseyhightower/status/953638870888849408

Why should I use GitOps?

We think GitOps has several major advantages although it's no silver bullet that works in every case. It allows you to deploy faster and more often, gives you easy and fast error recovery for free, and makes deployments more secure and self-documenting.

Deploy Faster and More Often

Probably every Continuous Deployment technology promises to make deploying faster and allows you to deploy more often. The uniqueness of GitOps is that you don't have to adopt new tools for deploying your application while developing. Everything happens by utilizing the version control system you already use for developing the application.

When we say "high velocity" we mean that every product team can safely ship updates many times a day — deploy instantly, observe the results in real time, and use this feedback to roll forward or back.

Weaveworks[4]

Easy and Fast Error Recovery

Oh no! Your production environment is down! With GitOps you have a complete history of how your environment changed over time. This makes error recovery as easy as issuing a `git revert` and watching your environment being restored.

The Git record is then not just an audit log but also a transaction log. You can roll back & forth to any snapshot.

Alexis Richardson[5]

[4] https://www.weave.works/blog/gitops-high-velocity-cicd-for-kubernetes
[5] https://twitter.com/monadic/status/1002502644798238721

Secure Deployments

GitOps allows you to manage deployments completely from inside your environment. For that, your environment only needs access to your repository and image registry. That's it. You don't have to give your developers direct access to the environment.

> **kubectl is the new ssh. Limit access and only use it for deployments when better tooling is not available.**
>
> **Kelsey Hightower**[6]

Self-documenting Deployments

Have you ever SSH'd into a server and wondered what's running there? With GitOps, every change to any environment must happen through the repository. You can always check out the default branch and get a complete description of what is deployed where plus the complete history of every change ever made to the system.

Better yet, using Git to store complete descriptions of your deployed infrastructure allows everybody in your team to check out its evolution over time. With great commit messages everybody can reproduce the thought process of evolving infrastructure and find examples of how to set up new systems, too.

> **GitOps is the best thing since configuration as code. Git changed how we collaborate, but declarative configuration is the key to dealing with infrastructure at scale, and sets the stage for the next generation of management tools.**
>
> **Kelsey Hightower**[7]

[6] https://twitter.com/kelseyhightower/status/1070413458045202433
[7] https://twitter.com/kelseyhightower/status/1164192321891528704

How does GitOps work?

After telling you what GitOps is and why we like it so much, we'd like to explain how it actually works. Most of the core concepts stem from the Guide to GitOps[8] by Weaveworks, focused on leveraging Weaveworks products on Kubernetes.

Environment Repository

GitOps organizes the deployment process around code repositories as the central element. There are at least two repositories: the application repository and the environment repository. The application repository contains the source code of the application and, optionally, a Dockerfile for building a container. The environment repository contains all deployment manifests of the currently desired infrastructure for the target environment. It describes what applications and infrastructural services (i.e., message broker, service mesh, monitoring tool) should run with what configuration and version. You could also leave the application deployment manifests in the application repository. We are discussing the advantages and disadvantages of this approach in the FAQ section: Where should I put deployment manifests?

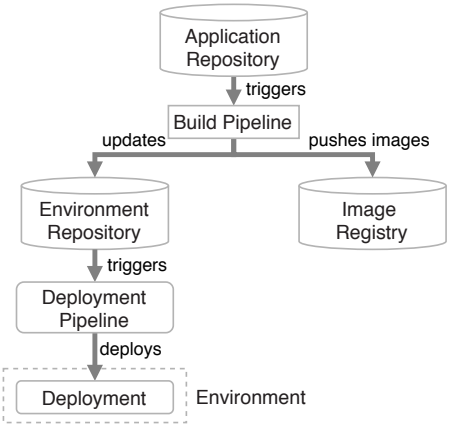
Push-based vs. Pull-based Deployments

There are two ways to implement the deployment strategy for GitOps: push-based and pull-based deployments. They differ in how you make sure that the target environment actually resembles the desired infrastructure. If possible, the pull-based approach should be preferred as it is considered the more secure and, thus, better practice to implement GitOps.

[8] https://www.weave.works/technologies/gitops/

Push-based Deployments

The push-based deployment strategy is implemented by popular CI/CD tools such as Jenkins[9], CircleCI[10], or Travis CI[11] — you may know it as CIOps. The source code of the application lives inside the application repository along with the Kubernetes YAML files needed to deploy the app. Whenever a developer updates the application code, the CI/CD system triggers the build pipeline, which builds the container images and updates the environment repository with new deployment manifests. It's common to store templates of the deployment manifests in the application repository, so the build pipeline can use these templates to generate the actual manifests, eventually stored in the environment repository.



Push-based Deployments

Changes to the environment repository trigger the deployment pipeline. This pipeline is responsible for applying all manifests in the environment repository to the infrastructure. With this approach it is indispensable to provide credentials to the deployment environment, which means that the pipeline has god-mode enabled. A

[9] https://jenkins.io/
[10] https://circleci.com/
[11] https://travis-ci.org/

compromised CI/CD system could act as a gateway to your target environment, with all ssh keys and secrets provided on a silver platter. In some use cases a push-based deployment is inevitable when running an automated provisioning of cloud infrastructure. In such cases, it is strongly recommended using the configurable authorization system of the cloud provider to restrict deployment permissions.

Also, to keep in mind that using this approach, the deployment pipeline is triggered *only* by changes to the environment repository. The system won't notice any deviation from the environment and its desired state automatically. Consequently, you might want to add some form of monitoring so that someone can intervene in case the environment doesn't match with the descriptions in the environment repository.

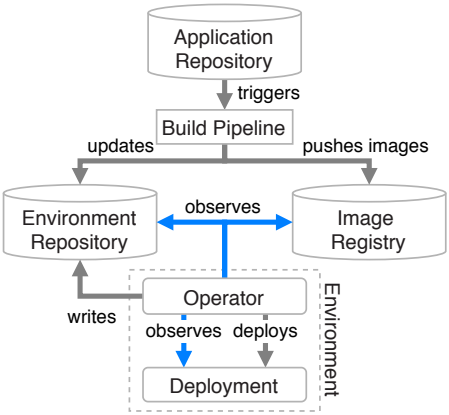
Want to see how to set it up? Check out Google's Tutorial[12] on how to set up push-based deployments with their Cloud Builds and GKE.

Pull-based Deployments

The pull-based deployment strategy uses the same concepts as the push-based variant but differs in how the deployment pipeline works. Traditional CI/CD pipelines are triggered by an external event, for example when new code is pushed to an application repository. With the pull-based deployment approach, the *operator* is introduced. It takes control over the pipeline by continuously comparing the desired state as defined in the environment repository with the actual state deployed in the infrastructure. Whenever the operator notices any differences, it updates the infrastructure to match the descriptions in the environment repository. Additionally, the image registry could be monitored to find new versions of images to deploy — but this depends on your Continuous Delivery workflow.

Just like the push-based deployment, this variant updates the environment whenever the environment repository changes. However, with a respective operator, changes can also be noticed in the other direction. Whenever the deployed infrastructure changes in any way not described in the environment repository, these changes can be rolled back to the respective configuration in the environment repository.

[12] https://cloud.google.com/kubernetes-engine/docs/tutorials/gitops-cloud-build



Pull-based Deployments

We generally recommend preventing manual changes to a production environment, since such undocumented alterations are known to be error prone and tricky to debug. By treating the environment repository as the single source of truth and preventing direct changes to the cluster, all changes become traceable in the Git log.

This is discussed quite controversially. The folks at Codefresh believe that this is a huge disadvantage[13] since dynamic changes of resources triggered by controllers are being blocked to be dynamic. Think of auto-scaling and resource optimization controllers which optimize workloads' scaling properties according to their demand. Fortunately, platforms such as Kubernetes allow discarding further information about dynamic parts of the configuration so such optimizations are not impacted when using static environment descriptions, where they are needed. Nevertheless, there is still room for improvement to tackle this issue.

Most operators also support sending emails or Slack notifications if they can't bring the environment to the desired state for any reason. For example, if they can't pull a container image. Additionally, you should probably set up monitoring for the operator itself, as there is no longer any automated deployment process without it.

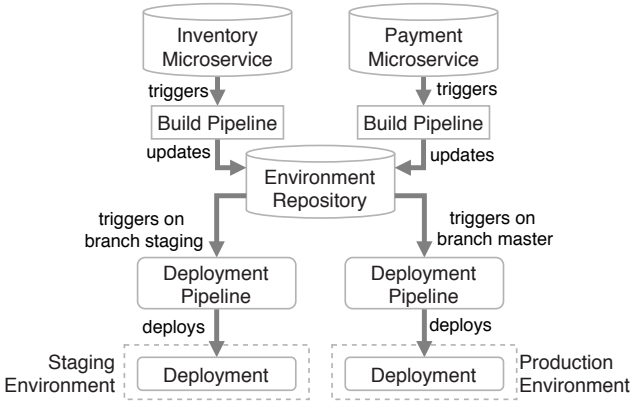
[13] https://codefresh.io/gitops-guide/

The operator should always live in the same environment or cluster as the application to be deployed. This prevents the god-mode of the push-based approach, where credentials for doing deployments are known by the CI/CD pipeline. When the actual deploying instance lives inside the very same environment, no credentials need to be exposed to external services. The authorization mechanism of the deployment platform in use can be utilized to restrict the permissions on performing deployments. This has a huge impact in terms of security. When using Kubernetes, RBAC configurations and service accounts can be utilized.

Working with Multiple Applications and Environments

Working with just one application repository and only one deployment environment is certainly not realistic for most systems. When using a microservice architecture, you probably want to keep each service in its own repository.

GitOps can also handle this use case. You can always just set up multiple build pipelines that lead to an update in the environment repository. From there on the regular automated GitOps workflow kicks in and deploys all parts of the application.



Multiple applications and environments

Managing multiple environments with GitOps can be done in several ways. You can think of one configuration repository for each environment: Dev, Staging, Production. Another possibility is to only maintain one repository for all environments with different directories for each environment. You could also split the environments with branches in one environment repository. This way, a promotion is done by merging changes from one environment branch to another. For example, Dev environment changes can be propagated to the Staging environment by pull request, where the environment branch for Dev is still remaining and, thus, long-running. Such an approach ensures an audit-friendly and transparent process at one place with Git tools and peer review. You can set up the operator or the deployment pipeline to react to changes on one branch by deploying to the production environment and another to deploy to staging.

Preview Environments

Alongside the term GitOps another special practice emerged: preview environments, or sometimes also known as preview deployments. Although GitOps and preview environments share some concepts they are quite different in nature.

Let's say your team is currently working on introducing a new feature. You're almost finished implementing this feature, but you want to try it out in a production-like environment. Also, it is crucial for you to get feedback from someone in the UX and QA team before the feature goes into production. Tired of waiting for the sandbox environment provided by your ops team? That's where preview environments come in.

With preview environments, each code change automatically results in a deployment after all automated tests have passed. You get a ready-to-use but short-living production-like deployment to gather feedback early. Hightower et al. suggest deploying and testing an application during development for every commit. (Hightower, K., Burns, B., & Beda, J. (2017). Kubernetes: Up and Running: Dive Into the Future of Infrastructure. Sebastopol: O'Reilly, p. 10)

Again, there are two forms of preview environments: branch-based and pull request-based. The branch-based preview environments tie their lifecycle and unique URL

to a branch, whereas the pull request-based ones to a pull request. The creation of a branch or pull request creates a new preview environment. The update of a branch or pull request updates their corresponding preview environment, and the deletion destroys them. Gitlab-CI supports branch-based preview environments with their 'Review Apps'[14] and Jenkins X supports pull request-based preview environments with their very own 'preview environments'[15]. The pull request-based approach, however, has a great advantage: there's a place to put peer reviews and notifications. For example, after every successful deployment of a preview environment, Jenkins X adds a comment to the pull request. But be aware that developers make mistakes. It's possible that with preview environments, mistakes such as memory leaks are automatically deployed in your cluster. So be sure to add the possibility to limit or delete a deployment explicitly if it runs havoc.

Trunk-based Development vs. Pull Requests

Weaveworks advocates for operation by pull request[16]. Any operational change, including every deployment, is a result of a pull request getting merged. As the rollout of such a change is automated, the onboarding of new developers only requires permission to the respective Git repository and permission to merge pull requests. This approach ensures peer review and audit trails to make changes observable and accountable. Tools such as Snyk and Jenkins X use pull requests to propose configuration changes. As a vulnerability scanner, Snyk proposes mitigation of dependency vulnerabilities in this way to be reviewed and merged by a repository maintainer. Jenkins X is a CI/CD solution that implements GitOps in a highly opinionated manner. It uses pull requests to change deployment configurations and to trigger the deployment pipeline. After a successful deployment, Jenkins X automatically merges the corresponding pull request.

But how does this workflow fit into the common practice of trunk-based development for truly living Continuous Integration, preached by highly valued opinionators such as Jez Humble and Martin Fowler?

[14] https://docs.gitlab.com/ee/ci/review_apps
[15] https://jenkins-x.io/v3/admin/guides/preview-environments
[16] https://www.weave.works/blog/gitops-operations-by-pull-request

There is a common misunderstanding when talking about trunk-based development. Branches aren't always evil. As long as the used branch is short-lived and maintained by one person only[17], branches are considered fine. So we can still do trunk-based development and utilize the power of pull requests for code review and build automation purposes. Even better, we can run our automated tests early on[18] to find issues much faster. This allows taking advantage of both, trunk-based development and pull requests in Git workflows.

[17] https://trunkbaseddevelopment.com/#scaled-trunk-based-development
[18] https://trunkbaseddevelopment.com/continuous-integration/#ci-pre-or-post-commit

GitOps vs. ...

Naming is hard – we all know that. With the rise of a new term like GitOps, you typically wonder whether it's something new or simply old wine in new bottles. We want to show you that GitOps is new wine in recycled bottles (we do care about the environment).

DevOps

DevOps is all about the cultural change in an organization to make people work better together. GitOps, on the other side, is a technique to implement Continuous Deployment. Although DevOps and GitOps share principles like automation and self-serviced infrastructure, comparing them doesn't really make sense. That being said, those shared principles certainly make it easier to adopt a GitOps workflow when you are already actively employing DevOps techniques.

BizOps

BizOps can be seen as another evolution of DevOps, namely, a cultural change with the business outcome as the primary measure of success as described in the BizOps Manifesto[19]. To some extent, DevOps already includes part of BizOps because DevOps stems from the lean movement and, thus, was never meant to only optimize the collaboration between developers and operations. The goal of DevOps was also to promote cross-functional teams and empathy across all departments of an organization while building software. Putting BizOps into perspective, we can say that both, GitOps and DevOps, are practices that can help achieve a better business outcome when applied pragmatically, and, therefore, can be seen as drivers for teams adopting BizOps.

NoOps

NoOps is all about extensive automation so that there's little to no manual operations work left to do. You can use GitOps to implement NoOps, but it doesn't automatically

[19] https://www.bizopsmanifesto.org/

make all operations tasks obsolete. If you are using cloud resources anyway, GitOps can be used to automate those. Typically, however, some part of the infrastructure like the network configuration, or the Kubernetes cluster you use is managed centrally by some operations team, and not by yourself. So operations never disappear magically.

CIOps

Your usual CI/CD pipeline builds an application, runs the tests, and, as a last step, deploys it. It's so common that Ilya Dmitrichenko from Weaveworks came up with a name[20] for it: CIOps. Basically every CI/CD system supports this flow natively.

Although CIOps is easy to set up, it has its limitations. For one, the CI/CD system needs to know credentials for the target environment. Furthermore, every application needs its own pipeline and there's no central place to see the deployment configuration for the whole system.

You could summarize CIOps as a technique that generally follows the GitOps push-based deployment strategy without making use of an environment repository.

SVNOps

Is there even SVNOps? In a way, yes. In principle, you can use any version control system you want to implement a GitOps workflow for. One of the core ideas of GitOps is letting developers use the tools they are familiar with to operate the infrastructure. If you prefer SVN over Git, that's cool! However, you may need to put more effort into finding tools that work for you or even write your own. All available operators only work with Git repositories — sorry!

[20] https://www.weave.works/blog/kubernetes-anti-patterns-let-s-do-gitops-not-ciops

FAQ

Is my project ready for GitOps?

Most likely: Yes! The cool thing about GitOps is that you don't need to write any code differently. What you need to get started is infrastructure that can be managed with declarative Infrastructure as Code tools.

How do I integrate GitOps into an existing CI/CD pipeline?

You are probably following the CIOps approach, where you build, test, and deploy your application in a push-based fashion in a single CI/CD pipeline. Fear not, you can gradually transition to a pull-based GitOps workflow in a few steps.

Normally, the deployment configurations are part of the application repository. So at first, you need to separate the deployment configuration from the application itself, hence, move them to the separate environment repository. Now, you need to move the CD part of the CI/CD pipeline from the application repository to the environment repository. The CI pipeline of the application repository should only run the tests, build the container image, and push the image to the container registry. The CD pipeline of the environment repository should only deploy the application.

However, somehow you need to set up a trigger for the CD pipeline to run. There are various possibilities to do this. You can do this manually through a user-triggered button click or in some automated form; for example, the CI pipeline triggers the CD pipeline when it's done. All of them are fine, just choose the one most suited for you.

You might wonder how the deployment manifests are changed to reflect a new container image version tag. You could do this manually in the deployment manifest, which then triggers the CD pipeline. But if you are truly doing Continuous Deployment, you might want to automate that, as it is only changing one property in a manifest. Nevertheless, if some breaking changes of the infrastructure configuration needs to be applied, you must do that over in the environment repository manually.

Congratulations, you do now have a push-based deployment strategy with an environment repository. You are, however, only half way there.

Conceptually, you now need to move the steps to deploy the application from the CD pipeline that's living outside our target environment into our target environment. The most straight-forward approach would be to literally move the CD pipeline into your target environment and set up an event-based trigger on pushes to the environment repository. If you're using Kubernetes you can use one of the available GitOps operators (e.g., Flux or Argo CD) available that do the same thing. In that case, the GitOps operator observes the environment repository and starts the deployment process when it detects a change.

I don't use Kubernetes. Can I still use GitOps?

Yes! GitOps is not limited to Kubernetes. In principle, you can use any infrastructure that can be observed, described declaratively, and has Infrastructure as Code tools available. However, currently most operators for pull-based GitOps are implemented with Kubernetes in mind.

Is GitOps just versioned Infrastructure as Code?

No. Declarative Infrastructure as Code plays a huge role for implementing GitOps, but it's not just that. GitOps takes the whole ecosystem and tooling around Git and applies it to infrastructure. With that, you gain all the benefits of code reviews, pull requests, and comments on changes for your infrastructure.

However, the biggest difference is the GitOps Operator, which lives in the target environment to deploy your applications securely. Those Continuous Deployment systems also guarantee that the currently desired state of the infrastructure is deployed in the production environment.

How to get secrets into the environment without storing them in Git?

First of all, never store secrets in plain text in Git! Never!

That being said, you could have secrets created within the environment. Such secrets never leave the environment. They stay unknown to everyone except the applications that need them. For example, you could provision a database within the environment

and give the credentials only to the applications interacting with the database. Use secret management systems for this such as HashiCorp's Vault[21].

Another approach makes use of a public/private key pair. First, you need to add the private key to the environment. Or more realistically, have someone from your dedicated ops team add the private key. Next, you can encrypt your secrets with your public key and add them to the environment repository by yourself. The environment can decrypt those secrets using the private key when necessary. There's tool support for such sealed secrets[22] in the Kubernetes ecosystem.

How does 'Operations by Pull Request' comply to Continuous Deployment?

We know what you are thinking. These people don't know the difference between Continuous Delivery and Continuous Deployment. If they knew, they would see that 'Operations by Pull Request' introduces a quality gate to every deployment, so it *cannot be* Continuous Deployment.

Yes, we know that! And it is a wonderful example on how flexible the GitOps concept can be! Let's say you are doing automatic deployments straight after your CI Pipeline turns green — without any manual intervention. You can totally do that with GitOps, since pull requests can be created and closed automatically over APIs without any manual approval. Just make sure to run your linting and infrastructure tests on every pull request of the environment repository, too, so you are safe to deploy automatically after closing that pull request with a merge. If you are leveraging Kubernetes for your deployments, we recommend using KinD[23] to automatically test your roll-outs before actually going to production.

How does GitOps handle DEV to PROD propagation?

GitOps doesn't provide a solution to promote changes from one stage to the next one. We recommend using only a single environment and avoiding stage propagation altogether. If you really need multiple stages (e.g., Dev, Staging, Production, etc.) with

[21] https://www.vaultproject.io/
[22] https://github.com/bitnami-labs/sealed-secrets
[23] https://kind.sigs.k8s.io/

a deployment environment for each, you could handle it by integrating changes from one stage to another using branches and merges. This also ensures a robust propagation process where it is transparent which state is deployed to which environment.

How do I trigger deployments from image registry push events?

If you leverage image registry events to update applications in your target environment update the deployment configurations in the environment repository accordingly. Please, don't bypass the repository and automatically update the new image directly in the target environment. Beware, we do GitOps to have full control and observability using the single source of truth — the environment repository, so version pinning is inevitable to achieve this. Those changes to the environment repository trigger the GitOps Operator to perform a new deployment, and the usual workflow kicks in.

Where should I put deployment manifests?

There are tools that allow the placement of manifest files inside the environment repository. The advantage of this is that infrastructure files are all in one place and can be reviewed by a dedicated platform team one pull request at a time. The disadvantage is, however, that you separate things that usually should be versioned together in one place.

An alternative approach is that the environment repository only collects a description of the target environment dependencies (aka umbrella manifest). This description acts as a link between your application repository and the environment repository, so the deployment operator knows where the deployment manifests are located to watch for. The downside of this approach is that you have commits (ideally automated ones) laying around in your repositories' Git history that change the desired version of your application in the respecting manifest. If you automate this as recommended, you need to deal with regularly pulling the current state of you repo, even if you are working alone on your project. This inconvenience can be prevented by storing your deployment manifests in a different place such as an object store. But by doing so you lose the versioning of your code together with your manifests changes that you would lose anyway if your manifests live in the environment repository in the first place.

I use Helm. Can I do GitOps then?

Absolutely! Every GitOps tooling explicitly supports Helm. Alternatively, you could render Kubernetes manifests out of your charts with the Helm CLI tool. So no reason to worry!

Should I hire GitOps engineers for my team now?

No! There are no GitOps engineers. GitOps is not a role (and neither is DevOps). GitOps is a set of practices. You can look for a developer who has experience practicing GitOps — or simply let your developers try out those practices.

GitOps Quickstart with the Flux Operator on Kubernetes

In this quickstart tutorial, we'll show you how to set up a pull-based GitOps workflow. We'll be using Flux[24], a Kubernetes-native GitOps operator developed by Weaveworks that comes with extensive documentation[25]. All code we use can be found in full on GitHub[26]. So if you feel stuck, just head over there to see the final result.

Prerequisites

To follow the tutorial, you'll need

- a GitHub account for two Git repositories and for running GitHub Workflows with Actions[27],
- a container registry for reading and writing container images,
- and a local Kubernetes cluster to run both, the GitOps operator and the application itself.

For the local Kubernetes cluster, we recommend using KinD[28], but Minikube[29], or Microk8s[30] is likewise well. We know that there are also Kubernetes clusters from the cloud computing providers like Google, Microsoft, or Amazon. For the sake of this tutorial, however, a local installation of Kubernetes suffices.

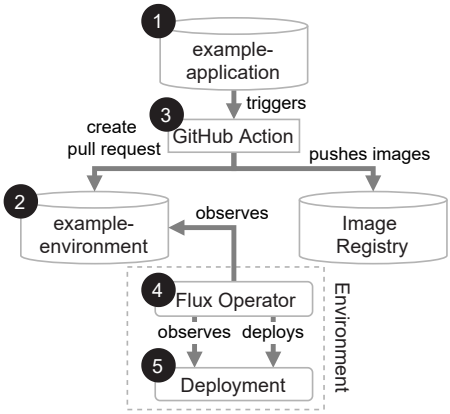
For the container registry, we recommend Docker Hub[31]. A free account at Docker Hub is more than enough for the sake of this tutorial. When you read this, the GitHub container registry might already be out of beta and a more convenient choice, though.

---

[24] https://www.weave.works/oss/flux
[25] https://toolkit.fluxcd.io/get-started/
[26] https://github.com/gitops-tech
[27] https://github.com/features/actions
[28] https://kind.sigs.k8s.io/
[29] https://minikube.sigs.k8s.io/docs/start/
[30] https://microk8s.io/
[31] https://hub.docker.com/

Big Picture

Okay, let's have a closer look at what we want to achieve. There are five major steps as shown in the figure below.



Our five steps to set up pull-based GitOps with Flux

First, we want to have an application repository for our example-application our developers can work on. Second, we want to have an environment repository for our infrastructure manifests. Third, we need to set up a GitHub Workflow to update the applications deployment manifest inside the environment repository via a pull request. Fourth, we need to deploy Flux to Kubernetes so that it observes the environment repository. And fifth, we want to kickstart a deployment via Flux as our GitOps Operator.

Step 1: Application Repository

The first step is setting up the application repository. We host our application repository at GitHub. The example application we'll be using in this tutorial is a simple web application written in Go that responds to requests with "Hello World". We creatively name the application repository `example-application` and add the following `main.go` and `Dockerfile` to it.

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello World.")
    })
    log.Fatalf("error: %s", http.ListenAndServe(":8080", nil))
}
```

main.go

```
FROM golang:1.13 as builder
ENV CGO_ENABLED=0
WORKDIR /app
COPY . /app/
RUN go build -o go-app

FROM scratch
ENTRYPOINT ["/go-app"]
COPY --from=builder /app/go-app /
```

Dockerfile

After you've created those files, your application repository should look like this.

```
.
├── Dockerfile
└── main.go
```

Directory structure of the application repository

Step 2: Environment Repository

Let's create the environment repository, we name it `example-environment`. See, naming can be easy sometimes. Next, we create a new directory named `applications` and place a subdirectory in it called `example-application`, the name of the application repository. This is the place where we put the Kubernetes manifests for our application.

Instead of placing the plain Kubernetes manifests in the environment repository, we could have used Helm or Kustomize as well. For simplicity we stick to the standard Kubernetes manifest format here.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-application
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-application
  template:
    metadata:
      labels:
        app: example-application
    spec:
      containers:
      - name: example-application
        image: <image-name>:<tag>
        ports:
        - containerPort: 8080
```

applications/example-application/deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: example-application
  labels:
    app: example-application
spec:
  type: ClusterIP
  selector:
    app: example-application
```

```
  ports:
  - port: 8080
```

applications/example-application/service.yaml

After you've created those files, your environment repository should look like this.

```
.
└── applications
    └── example-application
        ├── deployment.yaml
        └── service.yaml
```

Directory structure of the environment repository

Step 3: Continuous Delivery Pipeline with GitHub Actions

The job of our Continuous Delivery pipeline is straightforward: On every release in the `example-application` repository, build the container image, push the container image to the container registry, and create a pull request to update the deployment manifests to the new container image in the `example-environment` repository. We plan to automate all of this through a GitHub Workflow.

Instead of writing this from scratch, copy the `.github/workflows/newRelease.yaml` workflow file from our example application repository[32] in yours. GitHub automatically detects all workflows in the folder `.github/workflows`. This workflow runs two jobs on every new release: the *build* and the *release* job.

The build job creates the container image using the `Dockerfile` and pushes it into the default container registry, in this case Docker Hub. The job, however, needs credentials to push a container image to the registry. That's why we need to create two secrets in the GitHub example-application repository settings:

- `REGISTRY_USER` is the username you use at Docker Hub
- `REGISTRY_TOKEN` is an access token you can create to authenticate at Docker Hub

The release job runs only after the build job successfully pushed the container image to the registry. It's goal is to trigger a GitHub Workflow in the `example-environment` repository, which in turn will create the desired pull request. Think of this as an API call from one GitHub Workflow to another. However, the trigger for the environment repository is without effect because there is no GitHub workflow with the name "New Application Version" to trigger, yet. But fear not, we'll create that missing workflow now. To do so, copy the GitHub Workflow definition `.github/workflows/newApplicationVersion.yml` from the example environment repository[33] in your environment repository. This one is a bit large, but essentially it takes the arguments from the trigger and updates the deployment manifests (read: the image tag) accordingly within a pull request in a separate branch.

[32] https://github.com/gitops-tech/example-application
[33] https://github.com/gitops-tech/example-environment

At last, for your pipeline to actually work, you need to generate a personal access token with the scope `repo`, and place it as a secret named `PERSONAL_ACCESS_TOKEN` for both repositories. This is necessary so that the first workflow can trigger the second one as well as that the second one can create a pull request.

Let's try it out! Create a tag with the name `v1` in your local `example-application` repository and push it:

```
$ git tag --annotate v1 --message "Release v1"
$ git push
```

Observe the GitHub Workflow run in your repository on GitHub under the tab `Actions`. When the `build` step finishes, a new container image with the tag `v1` was published at Docker Hub. Go see it yourself! When the `release` step successfully finished, go to your environment repository and have a look at the workflow that has been triggered automatically over there as well. When this step finished, a new pull request should have popped up in the environment repository, including a change in the deployment.yaml that reflects the new container image tag. Magical! Merge this pull request, so we can deploy this application later on.

Congratulations, you set up an *Operations on Pull Request* workflow, which is already GitOps!

However, we did not deploy anything yet, which we will do in the next step.

Step 4: Flux Installation

The next thing we need to do is to deploy Flux into the Kubernetes cluster. Since Flux v2, however, Flux is not a single operator anymore. Instead, Flux basically comprises many operators, components in the language of Flux. But because we can deploy those components together, you can still view it as a single thing for the sake of this tutorial. The idea is that Flux continuously observes the environment repository, and whenever a manifest changes, Flux immediately applies the updated manifests in the cluster. It is also possible that Flux monitors changes of deployment manifests living in the respective application repositories. In this tutorial, we will use the possibility to store all manifests in one place: the environment repository. If you would like to know about advantages and disadvantages on putting your manifest files into the environment or application repository, read this section of the FAQ: Where should I put deployment manifests?

Before we can continue, please install the Flux CLI[34]. Although you could generate all required manifests with the Custom Resources manually and apply them to the cluster through `kubectl` by yourself, it's not advisable. The Flux CLI makes it so much easier to quickly generate all needed files. That's why we use it. Furthermore, it comes with a handy feature as well: you can check whether the Kubernetes Cluster suffices the needs of running Flux via: `flux check --pre`.

So let's start. We want to deploy the Flux components in our cluster with the `flux bootstrap` command. This command is very powerful as it does most of the bootstrap work. The only thing for you to do beforehand is to create a personal access token at GitHub for the `repo` scope — specifically for Flux. To pass the token securely to the command, we save it in our local shell like this: `export GITHUB_TOKEN=< your-token >`. Because we care about security, forgive us for this plead: Please do not reuse your tokens, even when multiple ones grant the same permissions as you should be able to revoke these tokens for each tool or use case separately. One last thing: just make sure to replace `$GITHUB_USER` with your GitHub user in the command below before you execute it.

[34] https://toolkit.fluxcd.io/get-started/#install-the-flux-cli

```
$ flux bootstrap github \
  --owner=$GITHUB_USER \
  --repository=example-environment \
  --branch=main \
  --path=./cluster-config \
  --personal
```

The output of the `flux bootstrap` command should look like this:

```
► connecting to github.com
/ repository cloned
+ generating manifests
/ components manifests pushed
► installing components in flux-system namespace
/ install completed
► configuring deploy key
/ deploy key configured
► generating sync manifests
/ sync manifests pushed
► applying sync manifests
◎ waiting for cluster sync
/ bootstrap finished
```

We now have a running instance of Flux operating your cluster. We can double check for the deployment of Flux to appear in the `flux-system` namespace, that was created automatically.

```
$ kubectl --namespace=flux-system get deployments
```

You might have noticed two things happening automatically now. First, Flux has generated a deploy key for your environment repository on your behalf — chill, this is fine. And second, because the folks from Weaveworks love *Infrastructure as Code*, Flux placed its own Kubernetes manifests inside the environment repository as well, namely, in the new subdirectory `./cluster-config/flux-system`. The file name prefix `gotk` indicates that Flux is using it's GitOps Toolkit[35] — a set of APIs and controllers, under the hood. Neat.

Let's have a closer look at the options we passed to the `flux bootstrap` command. The `github` argument specifies the code repository provider to be used. With the

[35] https://toolkit.fluxcd.io/components/

`path` option, we specified that only changes in the `cluster-config` directory of the environment repository trigger a pull-based deployment. If we had multiple environments, such as staging or dev we would either create new directories or a fresh new repository. This is your choice!

Step 5: Pull-based Deployment

Next, we want to deploy our application. We could make this quick and just tell you to throw your application's Kubernetes manifests somewhere in the `cluster-config` directory of the environment repository and we'd be done. Why? Well, Flux automatically applies anything within that directory automatically.

But we decided to show you the proper way. The way which increases security, observability and, thus, the ability to debug. The way that makes your setup future proof as you can follow the same steps even if you decide to put the deployment manifests not in the environment repository but in the application repository instead.

Let's dig into it.

For Flux to work, it needs to know

- where the Git repository with the deployment manifests of our application is,
- in which interval it should copy the entire repository to keep track of changes.

In our case, the Git repository with the deployment manifests is the `example-environment` repository and we use a 30 seconds pull interval. But first and foremost, Flux needs access to that Git repository. Because we host our environment repository privately at GitHub, we need to explicitly grant Flux access to it. For that we leverage the Flux CLI. The CLI is able to generate an SSH key and save it as a secret in Kubernetes — so we can later add it as a deploy key to GitHub. So let's generate it now.

```
$ flux create source git applications \
  --url=ssh://git@github.com/$GITHUB_USER/example-environment \
  --branch=main \
  --interval=30s \
  --namespace=default
```

You can see the key in the command's output. Now, you need to add it as a deploy key at GitHub in your environment repository's settings. Name this key `flux-./applications` as this key will work for all deployment manifests for all applications you are defining inside the environment repository. Leave the checkbox `Allow write access` unchecked, since Flux only needs to pull our application's manifests.

Next, as we want to follow Infrastructure as Code paradigm, we need to create a file that defines our custom resource `GitRepository`. It has the path `cluster-config/applications/applications-source.yaml`. In our case that location is still within the environment repository, but we could also have been pointing Flux to an application repository instead. To do so, run the same CLI command again, but this time with the additional option `--export > ./cluster-config/applications/applications-source.yaml` like so:

```
$ flux create source git applications \
  --url=ssh://git@github.com/$GITHUB_USER/example-environment \
  --branch=main \
  --interval=30s \
  --namespace=default \
  --export > ./cluster-config/applications/applications-source.yaml
```

The resulting manifest looks like this:

```
apiVersion: source.toolkit.fluxcd.io/v1beta1
kind: GitRepository
metadata:
  name: applications
  namespace: default
spec:
  interval: 30s
  ref:
    branch: main
  url: ssh://git@github.com/< your-github-username >/example-environment
```

cluster-config/applications/applications-source.yaml

Next, we create another manifest of the kind `Kustomization` explicitly for our application. It describes where Flux can look for changes of the application's manifests and in what interval it is doing that.

```
apiVersion: kustomize.toolkit.fluxcd.io/v1beta1
kind: Kustomization
metadata:
  name: example-application
  namespace: default
spec:
  interval: 5m0s
  path: ./applications/example-application
  prune: true
```

```
  sourceRef:
    kind: GitRepository
    name: applications
  validation: client
```

cluster-config/applications/example-application.yaml

Now, we could create these Kustomization manifests for each of our applications to deploy. For now, we only have this one. The directory structure of your environment repository should look like this:

```
.
├── applications
│   └── example-application
│       ├── deployment.yaml
│       └── service.yaml
└── cluster-config
    ├── applications
    │   ├── applications-source.yaml
    │   └── example-application.yaml
    └── flux-system
        └── [..]
```

Directory structure of the environment repository

Commit and push the new files.

It might, however, take some time until Flux actually deploys our application. This is because Flux monitors application manifest changes at a frequency of 5 minutes as configured in the `cluster-config/applications/example-application.yaml`. If you cannot wait you can fire up the following command:

```
$ flux reconcile source git applications
```

If there is a change in the default branch, the operator adjusts the cluster state. As you know, you should restrict the permission on altering deployments directly on the cluster using kubectl. But if it happens by accident, don't worry, Flux takes care of it and reverts any change made on the cluster to match the desired state described in your code repositories.

So let's have a look at the applications Flux is aware of as follows.

```
$ watch flux get kustomizations -A
NAMESPACE     NAME                 READY     MESSAGE
default       example-application  True      Applied revision: main/...
flux-system   flux-system          True      Applied revision: main/...
```

Great, we can see that our application is up and running. Before we can, however, access our deployed application locally, we need to set up a port forwarding rule as follows.

```
$ kubectl port-forward deployment/app 8080:8080
```

Now, we can finally get our "Hello World" response when calling our Go web application.

```
$ curl localhost:8080
```

Awesome! It always feels good to get this very first "Hello World" response when using a new technology.

Wrapping up the Tutorial

So let's summarize what we did in this tutorial. We've created two GitHub repositories, the application and the environment repository. We've installed and configured Flux to watch out for changes of the deployment manifests of our application. Since Flux observes the environment repository, a new deployment is triggered when a manifest change is merged into the main branch. This manifest update is made by two GitHub Workflows that act automatically and leverage operations on pull request.

If you want to get even more out of this tutorial, here are a few bonus steps.

- You can add another application repository and add this so Flux deploys it automatically as well. We suggest that you should create a Pirate version of the example application named named `pirate-application` which returns "Ahoy world"[36].
- You could make changes on your application directly in your cluster using kubectl e.g. by scaling it up or down. You should observe that Flux reverts this change. Also try to disable this behavior with the following command: `flux suspend kustomization <name>`. This could be helpful, for example, for debugging a deployment.
- You could set up a workflow with Flux where the application's deployment manifests live inside the application's repository. For that, you have to throw away the GitHub Workflow we set up and, instead, need to make sure that the manifests are updated for every new release by yourself!
- You could extend the GitHub Workflow of the environment repository to test your cluster setup with KinD[37]. This way, you are truly performing automated testing for your infrastructure configuration, too. Use this handy GitHub Action for it: KinD (Kubernetes in Docker) Action[38].

[36] https://lingojam.com/PirateSpeak
[37] https://kind.sigs.k8s.io/
[38] https://github.com/marketplace/actions/kind-kubernetes-in-docker-action

The Future of GitOps

The future of GitOps lights bright. In November 2020, the GitOps Working Group[39] was founded under the umbrella of the Cloud Native Computing Foundation (CNCF) by the leading companies behind GitOps, namely, Amazon, Codefresh, GitHub, Microsoft, and Weaveworks. They've started working on defining the GitOps principles[40]. With so many powerful players behind it, we expect both, the GitOps as a concept and the GitOps tools from the various vendors to evolve. So stay curious on what the future might bring. Just keep in mind that there are no silver bullets, and so GitOps isn't one either.

[39] https://github.com/gitops-working-group/gitops-working-group
[40] https://github.com/open-gitops/documents/blob/main/PRINCIPLES.md

Resources

So you've read our little book and are eager to try out GitOps and learn even more about it. To help you get started on your journey of becoming an expert in GitOps, we've curated a list of popular GitOps tools as well as articles and videos on GitOps. Happy learning!

GitOps Tools

- ArgoCD[41] is a GitOps operator for Kubernetes with a web interface.
- Flux[42] is the GitOps Kubernetes operator by the creators of GitOps — Weaveworks[43].
- Helm Operator[44] is an operator for using GitOps on K8s with Helm in combination with Flux.
- GitOps Engine[45] is a library for creating GitOps tooling for Kubernetes.
- GitOps Toolkit[46] is a set of APIs and controllers for creating GitOps tooling for Kubernetes.
- Gitkube[47] is a tool for building and deploying docker images on Kubernetes using `git push`.
- JenkinsX[48] is a Continuous Delivery solution on Kubernetes with built-in GitOps.
- Terragrunt[49] is a wrapper for Terraform[50] for keeping configurations DRY, and managing remote state.
- WKSctl[51] is a tool for Kubernetes cluster configuration management based on GitOps principles.
- werf[52] is a CLI tool to build images and deploy them to Kubernetes via push-based approach.
- codefresh[53] provides a GitOps automation platform for Kubernetes apps using ArgoCD.

Also check out Weavework's Awesome-GitOps[54] for even more pointers.

---

[41] https://argoproj.github.io/argo-cd/
[42] https://docs.fluxcd.io/
[43] https://www.weave.works/technologies/gitops/
[44] https://docs.fluxcd.io/projects/helm-operator/en/stable/
[45] https://github.com/argoproj/gitops-engine
[46] https://toolkit.fluxcd.io/components/
[47] https://gitkube.sh
[48] https://jenkins-x.io/
[49] https://terragrunt.gruntwork.io/
[50] https://www.terraform.io/
[51] https://github.com/weaveworks/wksctl
[52] https://werf.io/
[53] https://codefresh.io/
[54] https://github.com/weaveworks/awesome-gitops

Articles on GitOps

- An Inside Look at GitOps[55]
- GitOps - Operations by Pull Request[56]
- GitOps: What, Why, and How.[57]
- What Is GitOps and Why It Might Be The Next Big Thing for DevOps[58]
- What is GitOps Really?[59]
- Guide to GitOps[60]
- GitOps mit Helm und Kubernetes (German)[61]
- The Essentials of GitOps (DZone RefCard #339)[62]

[55] https://devops.com/an-inside-look-at-gitops/
[56] https://www.weave.works/blog/gitops-operations-by-pull-request
[57] https://www.reddit.com/r/kubernetes/comments/dc8bfd/gitops_what_why_and_how/
[58] https://thenewstack.io/what-is-gitops-and-why-it-might-be-the-next-big-thing-for-devops/
[59] https://www.weave.works/blog/what-is-gitops-really
[60] https://www.weave.works/technologies/gitops/
[61] https://www.doag.org/formes/pubfiles/11761447/06_2019-Java_aktuell-Bernd_Stuebinger_Florian_Heubeck-GitOps_mit_Helm_und_Kubernetes.pdf
[62] https://dzone.com/refcardz/the-essentials-of-gitops

Videos about GitOps

- GitOps - Operations by Pull Request [B] - Alexis Richardson, Weaveworks and William Denniss, Google[63]
- Tutorial: Hands-on Gitops - Brice Fernandes, Weaveworks[64]
- What is GitOps? Next level delivery with Flux and Kubernetes by Rafał Lewandowski[65]

[63] https://www.youtube.com/watch?v=BSqE2RqctNs
[64] https://www.youtube.com/watch?v=oSFTaAuOzsI
[65] https://www.youtube.com/watch?v=5zt-jzKHwX8

About the Authors

Florian Beetz

@fbeetz_

Florian is a software engineer at Lion5. He graduated from University of Bamberg in 2020 with a Master's degree in International Software Systems Science. He became interested in GitOps when Simon offered a seminar on Kubernetes as a guest lecturer. Apart from that, he is interested in cloud computing, clean code, and software engineering techniques. In his free time, he likes to go rock climbing in the mountains.

Anja Kammer

@innoq

Anja is a senior consultant at INNOQ and works in development teams most of the time. She creates cloud-native web applications, deployment automation, and CI/CD workflows in particular. Anja's focus in talks and articles is on the topics of DevOps and cloud infrastructure. In addition to her work as an IT consultant, she is a lecturer at the HTW Berlin University and teaches the course 'DevOps and Site Reliability Engineering'.

Dr. Simon Harrer

@simonharrer

Simon is a senior consultant at INNOQ. As part of a team doing remote mob programming[1], he fights everyday for simple solutions with domain-driven design, fitting architectures such as microservices or monoliths, and clean code in Java, Ruby, or even JavaScript. He co-authored the book Java by Comparison[2] that helps Java beginners to write cleaner code through before/after comparisons.

[1] https://www.remotemobprogramming.org
[2] https://java.by-comparison.com

The core idea of GitOps is having a Git repository that contains declarative descriptions of the infrastructure currently desired in the production environment and an automated process to make the production environment match the described state in the repository.

If you want to deploy a new application or update an existing one, you only need to update the repository – the automated process handles everything else.

It's like having cruise control for managing your applications in production.

innoq.com/gitops

INNOQ

978-3-9821126-8-8