



GitOps 2.0

THE FUTURE OF DEVOPS

Table of Contents

1. The Pains of GitOps 1.0

- 1.1. GitOps tools
- 1.2. Splitting CI and CD
- 1.3. Promotion of releases between environments
- 1.4. Modeling multi-environment configurations
- 1.5. Auto-scaling and dynamic resources
- 1.6. GitOps rollbacks
- 1.7. Observability
- 1.8. Auditing
- 1.9. Running at scale
- 1.10. GitOps and Helm
- 1.11. Continuous Deployment
- 1.12. Secrets management

2. A Vision of GitOps 2.0

- 2.1. Visibility into the whole software lifecycle
- 2.2. Observability and business metrics
- 2.3. Promotion among different environments
- 2.4. Achieving Continuous deployment and full Git automation
- 2.5. Built-in handling for rollbacks and secrets
- 2.6. Running GitOps at scale
- 2.7. Making the vision for GitOps 2.0 a reality

3. GitOps 2.0 with Codefresh & ArgoCD

- 3.1. Getting visibility into deployments
- 3.2. Deployment value comes from features and not Git hashes
- 3.3. The new Codefresh GitOps dashboard
- 3.4. Argo CD Integration
- 3.5. CI/CD pipelines with Codefresh
- 3.6. Getting started with GitOps 2.0

Introduction

GitOps is the practice of deploying applications by using Git Operations only (and not clicking UI buttons). The paradigm already existed in one form or another but was officially named as “[GitOps](#)” in [2017](#) by Weaveworks and has since become very popular for Kubernetes deployments.

The concepts behind GitOps are quite straight-forward:

- Infrastructure as Code: Git is always the source of truth on what happens in the system
- Code changes always go through an automated process
- Deployments, tests, and rollbacks controlled through Git flow
- Integration with secrets providers
- No hand-rolled deployments: If you want to change the state you need to perform a Git operation such as creating a commit or opening a pull request

Specifically for Kubernetes, GitOps says that you must not use `kubectl` to change the cluster state in an ad hoc manner. Instead, the desired state should be defined within GitOps tools at any moment. Those continuously compare the current state with the desired state to ensure the system is running like expected.

The most popular GitOps tools today are [Flux](#) and [ArgoCD](#). As several organizations are adopting GitOps tools, it is clear that a set of best practices are needed in several areas that are not currently covered by the existing tools.

In this guide, we will describe several gaps and issues in current GitOps tools, and the biggest challenges when adopting GitOps workflows. Next, we will look at ideal scenarios, the features that tools should provide, and the standards we aim to implement in what we defined as our vision for GitOps2.0.

We will explain a set of solutions both in theory (the vision) as well as in practice (our implementation).

The pains of GitOps 1.0

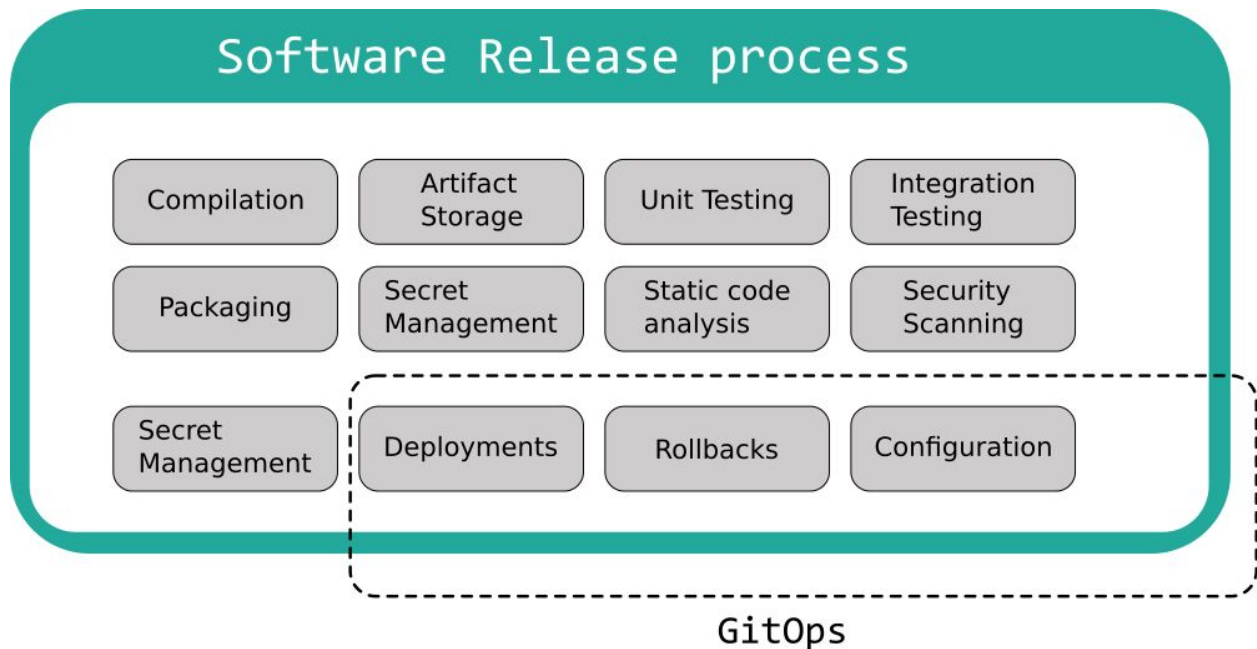
GitOps as a practice for releasing software has several advantages, but like all other solutions before it, has also several shortcomings. It seems that the honeymoon period is now over, and we can finally talk about the issues of GitOps (and the current generation of GitOps tools)

In the article we will see the following pain points of GitOps:

1. GitOps covers only a subset of the software lifecycle
2. Splitting CI and CD with GitOps is not straightforward
3. GitOps doesn't address promotion of releases between environments
4. There is no standard practice for modeling multi-environment configurations
5. GitOps breaks down with auto-scaling and dynamic resources
6. There is no standard practice for GitOps rollbacks
7. Observability for GitOps (and Git) is immature
8. Auditing is problematic despite having all information in Git
9. Running GitOps at scale is difficult
10. GitOps and Helm do not always work well together
11. Continuous Deployment and GitOps do not mix together
12. There is no standard practice for managing secrets

GitOps tools cover only a subset of the software lifecycle

This is the running theme of the current crop of GitOps tools. Even though GitOps (the methodology) has some interesting characteristics and selling points, the current GitOps tools focus only on the deployment part of an application and nothing else. They solve the “I want to put in my cluster what is described in Git” problem, but all other aspects of software development are NOT covered:



Gitops coverage

I am mentioning this because GitOps tools are sometimes marketed as the one-size-fits-all solution that will solve all your release problems and this is simply not true. First of all, GitOps requires that your deployment artifacts are already there. This means that tasks such as...

- Compiling code
- Running unit/integration tests
- Security scanning
- Static analysis

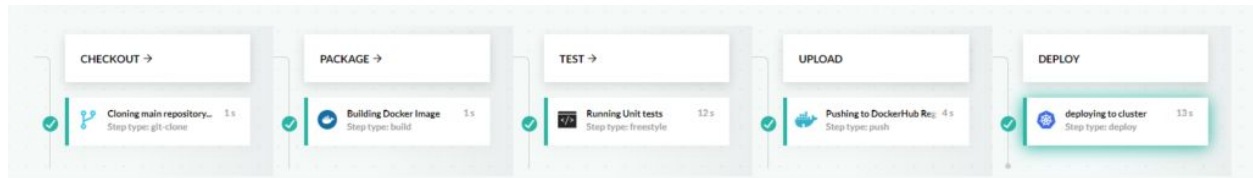
...are not a concern of GitOps tools and are assumed to already be in place.

Even several deployment concerns (such as promotion between environments, secret handling, smoke testing) are conveniently left out of the GitOps paradigm, and teams that adopt GitOps need to create their own best practices for all aspects of software delivery.

Therefore you cannot simply “adopt a GitOps solution” and call it a day. GitOps is only part of your whole development strategy and you should make sure that all other processes and workflows are ready to work with GitOps.

Splitting CI and CD with GitOps is not straightforward

GitOps has been heralded as a way to decouple CI from deployments. In the classic use of a CI/CD system, the last step in the pipeline is a deployment step.

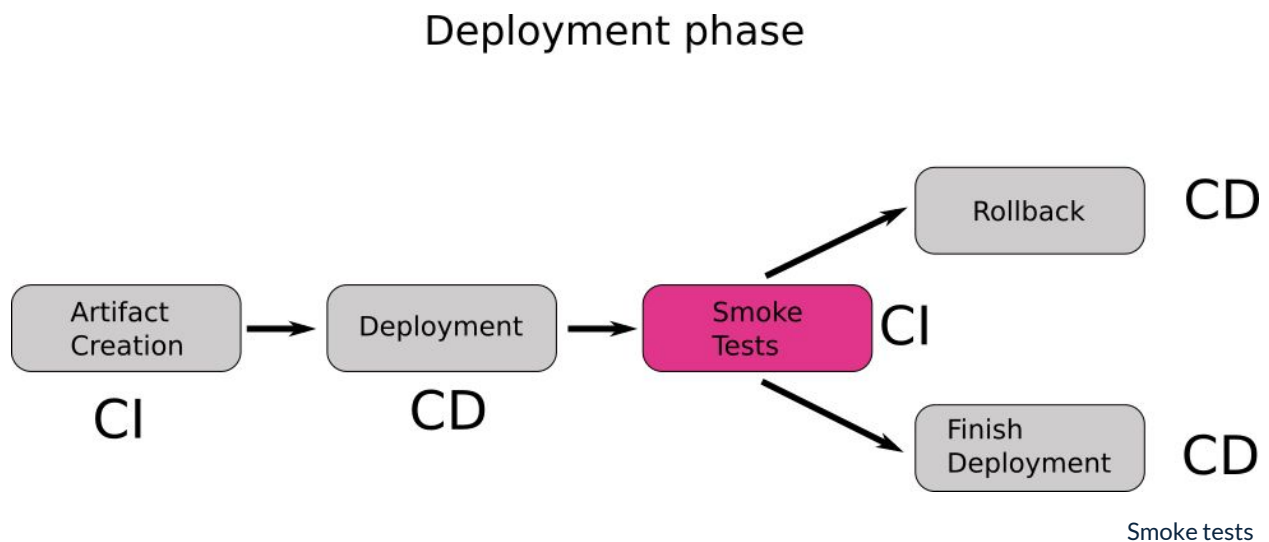


Classic pipeline

With GitOps you can keep your CI process pristine (by just preparing a candidate release) and end it with a Git commit instead of a deployment. The Git commit is picked up by a deployment solution that monitors the Git repository and takes care of the actual deployment by pulling changes in your cluster (and thus making the cluster state the same as the Git state).

This scenario is great in theory and is certainly applicable to simple scenarios, but it quickly breaks down when it comes to advanced deployments adopted by big organizations.

The canonical example of mixing CI and CD is with smoke testing. Let's say that you want to run some smoke tests AFTER a deployment has finished and the result of the tests will decide if a rollback will take place or not.



As I said in the previous point, GitOps deals only with deployment artifacts and normally does not touch (or know about) source code. But in most cases, in order to run unit tests, you need access to the source code of the application.

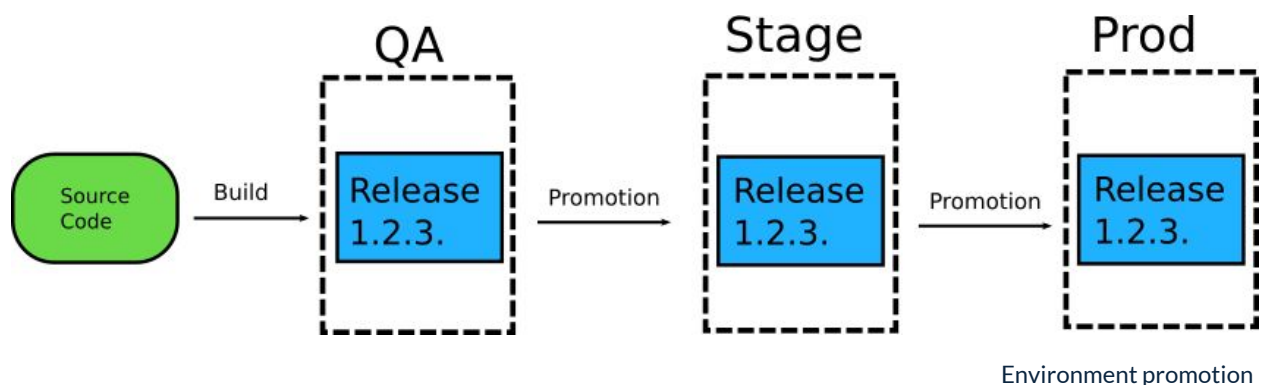
The current crop of GitOps tools cannot run unit/integration tests. That would require visibility in the source code along with all the testing frameworks and libraries needed for the tests. This means that you are forced to use your CI solution again in order to run the smoke tests.

The end result is a mixture of CI-CD-CI-CD components that goes against the main spirit of GitOps. There are also several underlying issues such as not knowing exactly when your environment has finished with the deployment in order to trigger the tests.

The same scenario is trivial to execute with a traditional CI/CD pipeline.

GitOps doesn't address promotion of releases between environments

This is probably one of the most well-known issues with GitOps and one of the first topics discussed when it comes to how GitOps can work in big organizations.



The basic scenario for one environment is obvious. You merge (or create a commit) in Git, and your cluster for environment X is now getting the new version. But how do you promote this release to environment Y?

Every time somebody declares that adopting GitOps is an easy process, I always ask how promotion between different environments works in their case. And I always get different answers:

1. “We use our CI system to do this.” This means that you are again mixing CI with CD and you admit that GitOps does not cover this scenario.
2. “We open a new pull request to the other environment.” This means that you are forced to have different Git branches for each environment (more on this later) and you also introduce further manual steps just for release promotion.
3. “We only have one environment.” Great for small companies, but not feasible in other cases.

I am really disappointed that even the [page specifically created for addressing GitOps questions](#) says:

“GitOps doesn’t provide a solution to propagating changes from one stage to the next one. We recommend using only a single environment and avoid stage propagation altogether.”

The most popular way of handling different environments seems to be by using different Git branches. This solution has several disadvantages:

- It opens the gates for people to do commits to specific branches and include environment-specific code.
- It makes your project coupled to specific environments (instead of being generic).
- It requires extra effort to keep all branches in sync (in case of hotfixes or configuration changes).
- It puts unnecessary strain on the CI system that has to check/rebuild/unit test each individual branch.

Also, if you have a large number of environments, handling multiple branches can get quickly out of hand.

There is no standard practice for modeling multi-environment configurations

A corollary to the previous point is that if your software strategy requires multi-environment deployments, GitOps cannot help you in any way. In fact, it will make things harder for you by forcing you to adopt a specific Git branching pattern (branch per environment).

A classic example is when you have different environments per geographical region (per continent or per country).

Let's say that my application is deployed to 10 countries and I want to promote a release to one after the other. What is the GitOps solution?

- A single repository with 10 branches. This means that you need to open/close 10 pull requests each time you do a release.
- 10 Git repositories. This means that you need to write your own solution that copies commits between the repositories (or uses Pull Requests between them).
- A single Git repository with 10 subfolders. Again you need an external solution to make sure that changes are propagated between folders.

In all cases, the promotion process is very cumbersome and current GitOps tools do not have an easy answer on what is the correct approach.

GitOps breaks down with auto-scaling and dynamic resources

One of the critical points in GitOps is that after a deployment has finished the cluster state is EXACTLY the same as what is described in the Git repository.

This is true in most simple cases, but as soon as you have dynamic values in your manifests, your GitOps tool will start fighting against you. Some classic examples are:

- The replica count if you have an autoscaler in your cluster
- The resource limits if you have an optimizer in your cluster
- Several other extra properties added by external tools (especially values with dates or timestamps)

As soon as your cluster state changes, your GitOps tool will try to sync the initial value from Git and in most cases, this is not what you want.

SUMMARYPARAMETERSMANIFESTDIFFEVENTS

☐ Compact diff
☐ Inline Diff

apps/Deployment/simple/simple-deployment

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app.kubernetes.io/instance: sample-deployment
6   name: simple-deployment
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: trivial-go-web-app
12  template:
13    metadata:
14      labels:
15        app: trivial-go-web-app
16    spec:
17      containers:
18        - image: 'docker.io/kostiscodefresh/simple-web-app:c88df35'
19          name: webserver-simple
20          ports:
21            - containerPort: 8080

```

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app.kubernetes.io/instance: sample-deployment
6   name: simple-deployment
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: trivial-go-web-app
12  template:
13    metadata:
14      labels:
15        app: trivial-go-web-app
16    spec:
17      containers:
18        - image: 'docker.io/kostiscodefresh/simple-web-app:c88df35'
19          name: webserver-simple
20          ports:
21            - containerPort: 8080

```

State difference

Argo supports custom diffs and Flux has a recommendation but I consider these workarounds as simple hacks that move away from the main GitOps promise and create several other issues in the long run.

There is no standard practice for GitOps rollbacks

The fact that all your cluster history is in Git makes rollbacks in GitOps (supposedly) very easy. If you want to rollback to a previous version, you simply use a past commit for your sync operation. In practice, however, it is not entirely clear what exactly we mean by “use a previous commit”. Different people use different ways to rollback:

1. You can simply point your cluster to a previous Git hash and let your GitOps tool sync that hash. This is the fastest way to rollback, but by definition leaves your cluster in an inconsistent state as the cluster does not contain what is described in the last Git commit.
2. You can use the standard Git reset, Git revert commands, and again let your GitOps tool perform the sync operation as usual. This keeps the GitOps promise (of having in the cluster what is in the Git repo) but of course, requires manual intervention.

3. You can have a combination where the GitOps tool itself both syncs a previous Git hash to the cluster and auto-commits (or reverts) to the git repo in order to keep the consistency. This is very complex to accomplish and not all teams want to let their deployment solution have write access to their Git repo.

It goes without saying that different people might want a completely different approach for rollbacks. At the time of writing, however, the present GitOps tools have very little support and guidance on how you perform a rollback in a standard way.

Observability for GitOps (and Git) is immature

GitOps is great for looking at your cluster state having the guarantee that it matches your Git state. Git hashes and commits however are only useful to developers and operators. Business stakeholders have no interest in which Git hash is now deployed at the cluster.

Therefore, while GitOps is great for observability on a technical level, it is important to remember that some of the most useful questions in a software team are the following:

- Does our production environment contain feature X?
- Has feature X cleared our staging environment?
- Are bugs X, Y present only in staging or also in production as well?

These kinds of questions are pretty important for most product owners and project managers and finding an answer should be as quick as possible.

Currently, GitOps tools work at the lowest level (i.e. Git hash) and don't have any connections to the business value of each deployment. It is up to developers/operators to find the correlation between a production deployment and the value it brings to the business.

At their current state, GitOps tools are great for observing the content of a cluster on a technical level but fail miserably in monitoring the business metrics of each deployment.

Auditing is problematic despite having all information in Git

A corollary to the previous point is that just because you have access to the whole deployment history of a cluster in the form of Git commits, doesn't mean that you can easily audit its functionality.

The current crop of Git tools are great for managing Git hashes but when it comes to searching and understanding business value, they can only provide simple free-text search capabilities as Adam mentioned already in [Lack of Visibility](#).

Let's say that you are using GitOps for a specific project and you know that the Git history matches your cluster history. How quickly can you answer the following questions just by looking at the Git History?

- How long did feature X stay in the staging environment before going to production?
- What is the worst, best, and average lead time (period starting from a developer performing a commit until the time it actually reached production) of the previous 2 months?
- What percentage of deployments to environment X were successful and what had to be rolled back?
- How many features exist in environment X but are not in environment Y yet?

These questions are very common in a large software team and unless you have a specialized tool on top of your deployment platform, it is very hard to answer them by only having access to a Git repository and its history.

Running GitOps at scale is difficult

This point was also touched by Adam in the part "[the proliferation of Git repositories](#)". If you adopt GitOps in a large company with a big number of environments and applications, the number of Git repositories quickly skyrockets.

This makes it very hard to keep track of what is going on in each environment and can quickly lead to configuration duplication or people making commits to specific environments (instead of using shared configuration).

For example, if you have 20 git repositories with Kubernetes manifests and you need to make a central change (e.g. adding a new company-wide label on each deployment) you need to manually make 20 Git commits or create some glue code that does it for you.

On the other end of the spectrum, you could have a single Git repository for all environments (or clusters) where all people collaborate along with the CI/CD systems.

This creates the problem of Git conflicts (where your Git repos are touched by many CI processes and pushes are failing because the git repository was changed in between), as already explained by Adam.

Having also a single gigantic Git repository can quickly become a bottleneck in your GitOps processes introducing performance issues when the repository is scanned for changes.

GitOps and Helm do not always work well together

Helm is the package manager for Kubernetes and is typically seen as the de-facto standard for deploying 3rd party applications in your cluster. It can also be used for your own deployments. Helm works by marrying a set of templates for Kubernetes manifests along with their runtime values that are merged to create the end result of what is deployed in the cluster.

While you can provide a values.yml file on the fly when a Helm release is installed, a best practice is to also commit the values file to Git. You can have different value values for each environment (e.g. QA/staging/production).

Helm on its own does not dictate where those 3 components (source code, manifests, values) should reside. You can keep all of them in the same Git repository or 3 different repositories. The accepted practice however is that if you have values for different environments, you don't keep them in the same place as the templates. The templates (the chart itself) is stored in a chart repository and the specific values for each environment are stored in a git repository.

This means that during a deployment the following must happen:

1. The chart should be downloaded from the chart repository
2. The values should be fetched from a Git repository
3. The values and the chart should be merged in order to create a set of Kubernetes manifests
4. The manifests should be applied to the cluster.

This is only for the initial deployment. Following the GitOps paradigm, the Git repository that contains the values file should be monitored along with the chart that exists in the chart repo. If their merge result is different from the cluster state a new deployment should take place. At the time of writing, neither [Flux](#) nor [ArgoCD](#) supports this basic Helm scenario. There are several workarounds and limitations that you have to accept if you wish to use GitOps with Helm charts making the process much more complex than needed.

Continuous Deployment and GitOps do not mix together

GitOps is great as a Continuous Delivery solution, where each commit results in a release candidate ready to be pushed in production. Continuous deployment on the other hand is the full journey of a commit straight into production without any human intervention.

You are practicing Continuous Deployment if as a developer you can commit something Friday afternoon and then immediately start your weekend. In a few minutes, your commit should land in production after it passes all quality gates and tests.

GitOps by definition is powered by Pull requests on a repository that contains your manifests. And in most cases, a human needs to look and approve these pull requests. This means that practicing GitOps involves at least some manual steps for handling these Pull requests.

In theory, one could practice Continuous Deployment while still adopting GitOps by fully automating the Pull request part. Current GitOps (and Git) tools are not however created with this automation in mind, so you are on your own if you want to take this route.

Remember that Continuous Deployment is not the holy grail. For several organizations, Continuous Delivery is already enough and in some cases, you might even be bound by legislation and have to specifically disallow fully automated deployments. But if you want to have the faster lead time possible, GitOps is not the best solution out there.

There is no standard practice for managing secrets with GitOps

This is a very well known problem with GitOps, so I am including it for completeness. Secret handling is one of the most important aspects of software deployment and yet, GitOps does not address them.

There is no single accepted practice on how secrets should be managed. If they are stored in Git, they need to be encrypted and thus have their own workflow process during a deployment. If they are not stored in Git, then the whole idea of having your cluster state the same as Git is not true anymore.

Secrets management is one of those areas where each company does their own thing, so at the very least I would expect GitOps tools to offer an out of the box solution for them.

A vision for GitOps 2.0

In the section above, we explained some of the issues we see with the current generation of GitOps tools (which we call GitOps 1.0). In this section, we will talk about the solution to those issues and what we expect from GitOps 2.0 – the next generation of GitOps tooling.

Visibility into the whole software lifecycle

The current generation of GitOps tools focuses only on the deployment part of an application artifact. The full software life cycle includes several other tasks until that point that deals with the packaging of the artifact, the unit tests, security scanning, etc. And even post-deployment there are several actions (such as running smoke tests) that are not covered by today's GitOps tools.

This means that if you decide to adopt GitOps, you already need to have an existing CI setup that works effectively and produces the artifacts that your GitOps tool will use. This approach works well for small setups, but having to look in two different places (your CI system for the artifact and your GitOps solution for the deployment) does not scale well for large organizations with multiple applications.

Ideally, you would want a single deployment platform that follows the full journey of each code change, from commit to deployment. This way you will have full traceability on each production change and can also easily answer critical DevOps metrics such as lead time (the time it takes from a commit to reach production).

Observability and business metrics

As an extension to the previous point, we believe that GitOps is a process that affects not just developers/operators but also the business stakeholders (i.e. product owners and project managers). Gaining visibility into the full software lifecycle is vital for everyone involved in software delivery and not just those who handle source code.

As we explained in the previous section (describing GitOps challenges) , some of the most important questions a software team should consider are:

- Does our production environment contain feature X?
- Has feature X cleared our staging environment?
- Are bugs X, Y only present in staging or also in production?
- What percentage of deployments to environment X were successful and what had to be rolled back?
- How many features exist in environment X but are not in environment Y yet?

These questions cannot be easily answered by GitOps tools today (in fact not even traditional non-GitOps deployment platforms can answer them). Having a GitOps solution in place that not only cares about Git hashes but also offers full visibility on the high-level metrics of each deployment will allow all stakeholders to monitor the deployment process and gain insights on what happens with each environment.

A very big gap today is the discrepancy between source code changes (i.e. Git commits) and business level features (i.e. issues from the ticket system such as JIRA) as they are typically handled by different systems. In order to create a correlation between them, developers often have to build additional cruft on top of the deployment platform and ever resort to custom low-level scripts that tie them together.

It is always important to remember that while on the surface a software project might seem like a series of code changes/commits, the visible value comes from business features that are getting shipped. Current GitOps tools can easily answer what Git hash is now deployed into a cluster, but more useful information like what features are present in that environment (and what features are waiting in another environment) is often missing or considered secondary knowledge.

Additionally, some of the classic DevOps metrics are very hard to monitor with current GitOps tools. Especially the lead time (the time it takes for a commit to reach production) cannot be measured easily if your deployment is handled by multiple tools and custom glue code (a very common occurrence in companies of all sizes).

It should be clear that we need dedicated tools for all these questions. Ideally, a GitOps solution should have native support for measuring and storing historical data for:

- Business features implemented in each release

- The journey of each software release from start to finish
- Correlation between git hashes, features, rollbacks, and environments
- Several deployment metrics (i.e. lead time)

Right now it is very hard to get all these features in a single overview.

Promotion among different environments

Promoting a released artifact between different environments is arguably the biggest challenge with GitOps tools right now. The model of using a Pull request which triggers a deployment once it is merged works great for a single environment but breaks completely when it comes to different environments.

Promotion of a release can take many forms, such as a linear progression to increasingly more critical environments (such as dev -> QA -> staging -> production) or as a parallel deployment to equally significant but slightly different environments (e.g. deploying a release to different geolocations). Neither scenario is covered sufficiently today by the current GitOps tools. Promotion between environments should be considered an integral part of any deployment solution. In the case of GitOps tooling, we can handle the promotion mechanism in a different number of ways, such as introducing an automation mechanism for Pull Requests (more on this in the next section) or even creating a higher-level abstraction layer for specifying different environments.

The end result should be that for any GitOps project a project stakeholder:

- clearly sees the releases present on current environments (including temporary dynamic environments)
- can easily move a release to the next environment (e.g. staging to production)
- can easily move a release to a previous environment (e.g. production to QA in order to examine a bug)
- has a configuration mechanism in place to define multiple version of the same environment (i.e. for different geographical regions)

Promotion of environments should always be possible to automate via a CLI (i.e. a graphical dashboard is not enough for all use cases).

Achieving Continuous deployment and full Git automation

As we explained in [our CI/CD/CDP article](#), Continuous Delivery is the process where each developer commit results in a release candidate that is ready to be sent to production. GitOps and Continuous Delivery are a perfect match, as the Pull Request process dictated by GitOps can function as a quality gate that approves/rejects the release candidate.

Continuous Deployment is the practice where every commit (that complies with the organization's requirements) is automatically sent to production without any human intervention. This is the ultimate form of a deployment pipeline, as it takes manual steps completely out of the equation.

Using Continuous Deployment and GitOps is an open question right now, as a Pull Request is for all intents and purposes a manual process. There is no best practice on how to practice Continuous Deployment using GitOps.

Pull Requests (like all other Git actions) can be fully automated. At the time of writing, there is no GitOps tool that does that. To achieve continuous deployment with GitOps we would need a GitOps solution which not only passively handles the opening and closing of Pull Requests but also opens/closes Pull Requests on its own in an unattended manner.

Automating Pull Requests might sound like an easy task, but in order to give plenty of visibility on what is happening at any point in time, we need a deployment solution that treats Pull Requests as a first-class citizen, instead of simply responding to Pull Request events like most tools do right now.

Built-in handling for rollbacks and secrets

We explained in the first point of this article that current GitOps tools focus only on the deployment part of the release cycle and ignore all other associated tasks. You could argue that this is their primary focus and they don't have to provide anything else regarding the software lifecycle.

Even if this was true, I consider secret management and automatic rollbacks an integral part of the deployment process. Unfortunately, both areas are currently left as an exercise to the reader if you try to adopt any of the existing GitOps tools.

We need a GitOps solution that handles secrets on its core offering. Secret management is also deeply connected with the problem of multi-environment installations that we already covered. A GitOps solution should not only offer a secure storage mechanism for all secrets, but also a comprehensive way on how to group them according to each environment and how to pass them in the respective cluster.

The underlying mechanism is not that critical. There is a common pattern that if you plan to use GitOps you must also store secrets in Git. I don't see this as an essential requirement for handling secrets in GitOps if there is a better proposal.

On the topic of rollbacks, the ideal GitOps tool should be able to perform the following:

1. Deploy the latest Git hash to the cluster
2. Get information from metrics regarding the success of the deployment
3. If metrics are ok, mark the deployment as finished
4. If metrics show failures, then automatically rollback to the previous version
5. Update the Git repository with the new information so that the latest Git commit matches again what is deployed in the cluster.

Today to achieve this workflow, you need to write custom glue code in addition to adopting a GitOps tool.

Running GitOps at scale

The last critical component of our vision for GitOps 2.0 is running GitOps at scale. Working with a small number of environments might seem easy, but in several real-world scenarios, the number of pipelines, projects, deployments, and environment configuration within a company can quickly get out of control.

Scalability is important not only in regard to technical capacity but also in regard to visibility and observability as explained in the previous sections. Different environments will exist at different

sync states, with multiple pull requests in flight and with multiple business features getting shipped.

We need a way to present all this information in both high-level and low-level representations. On one hand, we need a way to provide a bird's eye view of the whole company/department deployments to all stakeholders who need an easy and understandable dashboard for monitoring deployments along with their associated metrics.

On the other hand, anybody should be able to drill down and look at a specific deployment and the business value it contains, along with the exact status of each business feature it expects.

Making the vision for GitOps 2.0 a reality

In this part, we described a vision for the next generation of GitOps tools. We have also described extensively the gaps of the existing GitOps tools in the previous section.

At Codefresh, we believe that this vision is the next big thing for deployments and this is why we think it deserves the name of GitOps 2.0.

GitOps 2.0 with Codefresh and ArgoCD

In the previous section, we explained the vision behind GitOps 2.0 and the features we expect to be covered by GitOps 2.0 tools.

In this section, we will see how the [new Codefresh GitOps dashboard](#) is the first step towards this vision and more specifically in the area of observability and traceability.

Getting visibility into deployments

Deploying a new release into production is an important milestone for all project stakeholders. The developers write the code, the operators deploy the new artifact and product owners can finally mark a feature as “shipped”.

Current deployment tools however do not give the same attention to the needs of all team members that take part in a release. Most deployment tools today fall under the following categories:

1. Really low-level tools that are mostly useful to developers. They are full of build logs, Git hashes, and deployments events that mean nothing to product owners or business analysts.
2. High-level “deployment” tools that are actually a fancy UI layer on top of a low-level tool exposing only a minimal amount of information. These tools treat a deployment as a black box (as the actual deployment process is handled behind the scenes from another entity) and never give enough information on what went wrong if a deployment happened.

The reason behind this dichotomy is probably that each role needs access to different information. This might be true in a historical context but goes against the ideas and principles of DevOps (bringing all team members together).

Ideally, deploying a release should be so easy to perform/monitor/rollback that any team member can handle a deployment event and still have access to all critical information such as which Git commit was deployed and what new features are now available.

Deployment value comes from features and not Git hashes

In the previous section about the GitOps 2.0 vision, we outlined the gap that exists today between the low-level deployment information (i.e. the Git hash) and the value it brings after a deployment (i.e the actual features).

Product value is not measured in Git hashes, but instead in the number and significance of new features that are deployed to production. New features are typically recorded in an issue management system (e.g. JIRA) and have their own lifecycle according to what the organization desires.

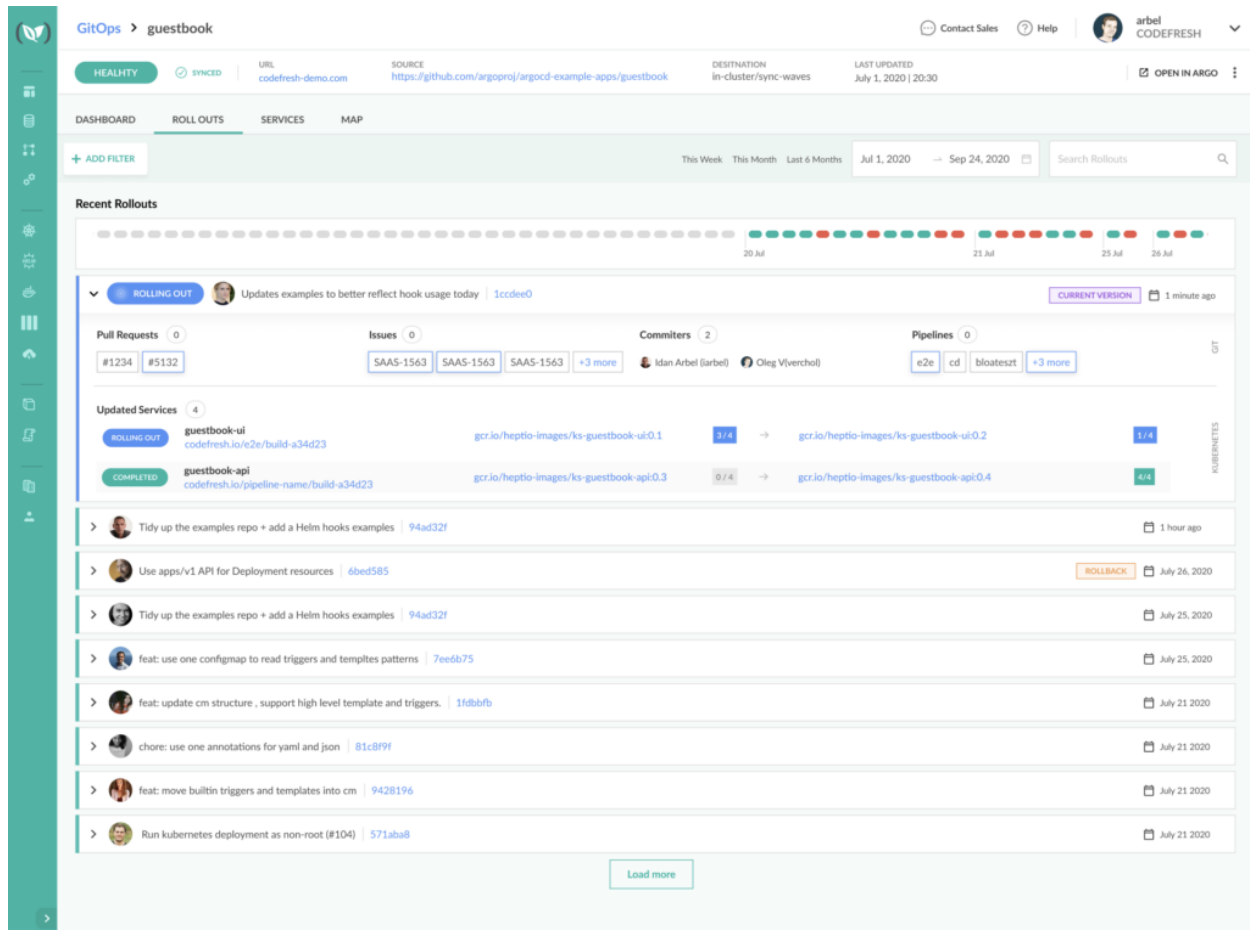
As a first step towards the vision of GitOps, we have decided to address this information gap and solve the observability problem by combining these two kinds of information (git hashes and features) in a single cohesive view.

You can now use the new Codefresh dashboard and correlate extra information on top of a single Git hash.

The new Codefresh GitOps dashboard

Our new GitOps dashboard provides a single pane where instead of looking for simple Git hashes, you can now see:

- The feature(s) implemented by this Git hash (i.e. JIRA issues)
- The Pull Request that was created for this feature
- The committer of the feature
- The timeline of deployment for this feature and if the deployment was successful or not
- The pipeline(s) that were used for building the container image and deploying it
- The Kubernetes images that were affected
- The Kubernetes services/deployments that were affected



Codefresh GitOps dashboard

With our new GitOps 2.0 dashboard, we are addressing the visibility and observability problems that plague existing GitOps tools.

Now you can finally get an overview of GitOps deployments in the context of features instead of just Git hashes. In addition, you can rollback to a specific feature (and/or pull request) in addition to a Git hash.

Using the dashboard you will be able to answer questions such as:

- Which feature was implemented by Pull Request 432?
- When was JIRA issue 5612 deployed in our QA environment?
- Is JIRA issue 2356 deployed in production now?
- Can we rollback the staging environment to just before JIRA-issue 2167 was shipped?
- Which pipelines were used for handling Pull Request number 156?
- Who pushed to our production (by committing into Git) last Friday afternoon?

- What JIRA features entered our staging environment this morning?
- What container images were rebuilt as a result of merging Pull Request 475?
- Which Kubernetes services were affected by JIRA issue 5923?

These are just a few examples of the capabilities you gain with the new GitOps 2.0 dashboard in its initial release. We have several other enhancements on our roadmap and we also welcome any additional feedback that you have regarding the structure and functionality of the dashboard.

Argo CD Integration

Instead of re-inventing the wheel, the new GitOps dashboard is powered by Argo CD, the popular deployment solution that can manage Kubernetes applications using a pull mechanism.

To take advantage of the Argo integration, you simply install our agent in the same namespace as the Argo services. Installation is as simple as running a single command via the Codefresh CLI:

```
kostis@cloudshell$ ./codefresh install argocd-agent
? Codefresh integration name argocd
? Argo host, example: https://example.com https://36.87.134.111/
? Argo username admin
? Argo password *****
Integration updated
? Select Kubernetes context gke_codefresh-support_us-central1-c_kostis-argo
? Select Kubernetes namespace argocd
argocd-agent "ClusterRole" already exists
argocd-agent "ClusterRoleBinding" already exists
argocd-agent "Deployment" already exists
argocd-agent "ServiceAccount" already exists
Argo agent installation finished successfully to namespace "argocd"
kostis@cloudshell$
```

Argo agent installation

Once the agent is installed, Codefresh has two-way communication with Argo CD via its API. For example, Codefresh will automatically mark an application as out-of-sync if the cluster configuration no longer matches the Git repository, which holds the manifests file that describes the application.

Environments > go-web-server

HEALTHY

⊗ OUT OF SYNC

CLUSTER

https://kubernetes.default.svc

NAMESPACE

simple

+ ADD FILTER

Out of sync

With the Argo CD integration in place, you can also define GitOps application right inside the Codefresh UI, with the same options available to Argo CD

Add ArgoCD Environment

Name
my argocd application

Connect to an existing application **Provision a new application**

Project Application
default my app

HUMAN **AUTOMATIC** Use schema to validate resource manifest ☒

SOURCE
Repository URL Revision Path
github:example/app HEAD goofbook

DESTINATION
Cluster Namespace
mydemo@kubernetes@BISpark.PL default

DIRECTORY
Directory Recursive ☐

CLOSE CREATE

Provision app

It is important to notice that at all times you can still use the Argo CD user interface in addition to the Codefresh UI.



Argo CD UI

All changes that you do in one place will be mirrored in the other.

CI/CD pipelines with Codefresh

With the Integration of Codefresh and Argo, you now have a full CI/CD solution at your disposal. Argo CD is a great deployment tool on its own, but it can only handle the last step in a pipeline created for GitOps deployment (the sync part).

A full deployment platform also needs the ability to compile applications, run unit tests, perform security scans, etc. These capabilities are out of scope for Argo, but are perfectly possible with Codefresh.

On the other hand, Codefresh already has strong CD foundations, but they are based on the push-model (deploying Helm packages on a cluster). With the addition of Argo, our customers can now create GitOps pipelines if they prefer.

This means that one can now create complete GitOps pipelines that are not possible by using Argo on its own.

For example, a very popular pattern for deployments is to have pre-sync and post-sync steps in a deployment pipeline.

The screenshot displays the Codefresh web interface for a pipeline named 'cd'. The pipeline is in a 'COMPLETED' state. It consists of three steps: 'PRE SYNC', 'SYNC APP', and 'POST SYNC'. The 'SYNC APP' step is currently selected, showing its detailed logs. The logs indicate a successful sync operation for the 'simple-kubernetes-deplo...' repository, commit 'P master/653402d', and pipeline 'cd'. The sync was performed on the HEAD (653402d) and the health status is 'Healthy'. The operation was a 'Sync' of revision '653402d34f41f6086b9b467f428e861ed7173bd8'. The phase 'Succeeded' started at '2020-09-10 12:32:44 +0000 UTC' and finished at '2020-09-10 12:32:45 +0000 UTC' with a duration of '1s'. The message states 'successfully synced (all tasks run)'. Below the main log, a table provides a summary of the synced resources:

GROUP	KIND	NAMESPACE	NAME	STATUS	HEALTH	HOOK	MESSAGE
Service	simple	simple	simple-service	Synced	Healthy		service/simple-service unchanged
apps	Deployment	simple	simple-deployment	Synced	Healthy		deployment.apps/simple-deployment configured

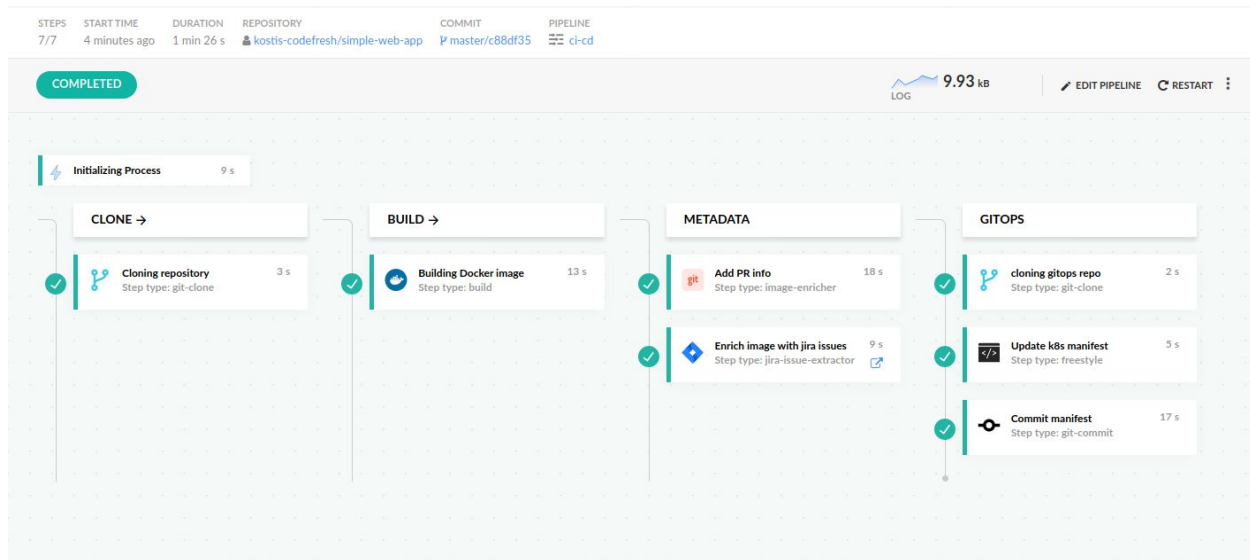
At the bottom of the log, it says 'Successfully ran freestyle step: ExecuteArgoCd'.

Sync pipeline

A possible scenario for a pre-sync step is to send a message to your monitoring solution that a deployment is about to take place. This way your monitor solution can place a “marker” on your time-based metrics to evaluate the health of the deployment (and possibly rollback automatically if the metrics are not healthy).

A possible scenario for a post-sync step is to perform smoke testing on the deployment that was just synced. Codefresh has great support for both unit and integration tests both of which require the capabilities of a full CI/CD platform.

As a more advanced scenario, Codefresh can actually orchestrate a single deployment pipeline that follows the GitOps paradigm as it is possible to automatically perform a Git commit inside a pipeline (as well as a replacement on the manifests).



Full GitOps pipeline

The example above is a single pipeline that:

1. Checks out the source code of the application
2. Builds a Docker image
3. Records the information for the current pull request what JIRA issue is solved
4. Creates a commit on the second git repository that holds only the Kubernetes manifests.

Of course, a pipeline like this can be easily enriched with additional steps for testing, linting, security scanning, etc.

It is important to note that the existing CD capabilities of Codefresh (and all dashboards such as [the Helm promotion one](#)) are still available to companies that wish to continue using them even if you don't want to follow the GitOps paradigm.

Getting started with GitOps 2.0

With the marriage between Argo and Codefresh, you get a [full CI/CD platform](#) that combines the ideas of GitOps, along with an extensible model of pipeline building that can match any possible deployment scenario.

To get started you can now open a [free account with Codefresh](#) and then follow our [documentation guide for GitOps deployments](#).