Book review Repeatability, Reliability, and Scalability through GitOps: Continuous delivery and deployment codified

By Bryan Feuling (Author)

Summary: A good book to show different ways and tools to archive good GitOps. It covers many tools benefits/drawbacks and logic/reasoning behind choosing it or not, e.g. helm, ArgoCD, JenkinsX, Jenkins, Flux, Circle CI, Terraform, Ansible, Pulumi, Harness. You can learn new and important GitOps concepts, e.g. Common industry practices for deployment/delivery, The Original/Purist/Verified GitOps, CD in Kubernetes, Serverless deployment-as-code, Declarative files overload, Best practices, etc.

First time see a book talking about new Harness which is a SaaS CD solution. It also helps you to understand many designs and user requirements. There are a number of useful tools (e.g. jsonnet) and useful Vscode extensions, e.g. Kubernetes, helm Intellisense, GitLens, etc.

There is a chapter 11 for GitOps pitfalls, e.g. how ArgoCD works and why/when sometimes out of sync because of possible issues. Also, important topics on Kubernetes manifests, failure strategies, governance and approvals, etc.

Suggestions: Some typos on pages 82 (“*tany”->”many”)*, p.84 (“*filesthe”-> “files the”)*, p.89 (“three car objects” -> “two car objects”), p.242 (Figure 11.38 – Self Heal should be “Disable” instead of “Enable”). I hope it can include GitLab in a future revision. Also, good to add more diagrams/illustrations in the first 8 chapters.

Details:

Section 1 (Ch. 1-3) is more on the fundamentals of GitOps, e.g. what and why GitOps, automation test, continuous deployment, delivery, pipeline, common practices, to push or pull.

Section 2 (Ch. 4-7) describes GitOps types, benefits and drawbacks. It talks about Continuous Deployment in Kubernetes, serverless deployment-as-code, manifest explosion, and common tools, e.g. ArgoCD, Flux, GitKube, and JenkinsX. I like this from p.53 most, *“There are some who would argue that the role of a Kubernetes administrator is not really a Kubernetes administrator, but more of a YAML engineer.”*

P.55 has a good discussions on ArgoCD drawbacks, e.g. manifest explosion, no cross-cluster deployments, originalist GitOps with ArgoCD is for deployments only.

p.67, “However, the generation or mutation of the resource manifests is where Jenkins shows significant gaps. Because Jenkins does not have a native build manifest option an administrator or developer would need to create a manifest template for Jenkins to manipulate.”

p.69, “Declarative files overload”

p.70, “By the end of the pipeline design process, the DevOps team figured that they would need to create and maintain at least 100 core files. But when they considered all the Terraform values files, the Helm Chart values files, and Jenkinsfiles, they figured that they would need to support between 300 and 500 declarative files in total. And to make matters worse, each file referenced at least one other file, with some referencing up to 15 different files. This meant that if there was an issue with a deployment, the troubleshooting process would be like trying to unravel an entire plate of spaghetti, without breaking a noodle.”

p.73,”The main drawback of a process such as purist GitOps is in the size and number of the required declarative files. If the declarative files are too large, then the configurability of each file becomes a usability issue. Alternatively, if the files remain small but the number of files grows, then tracking those files and file relationships becomes an administration issue.”

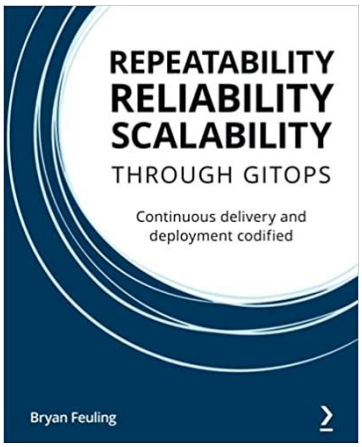
Chapter 6 - Verified GitOps – Continuous Delivery Declaratively Defined

p.77, “The essential difference between verified GitOps and the other GitOps practices is the desire to leverage GitOps for the entirety of continuous delivery, rather than only for continuous deployment.”

Section 3 (ch8-12) is the most important. It talks about HANDS-ON practical details.

1/ buy it at Amazon: https://www.amazon.com/Repeatability-Reliability-Scalability-through-GitOps/dp/1801077797

/ sample code in GitHub - https://github.com/PacktPublishing/Infrastructure-Monitoring-with-Amazon-CloudWatch



pull or push ?

With an architecture such as Kubernetes, the pull-based deployment method is significantly easier to administrate since permissions and network connectivity are typically configured for the cluster already. And since common practice with Kubernetes is to only have a handful of clusters, the cost of ownership is significantly lower. For push-based deployments, an external deployment engine would require additional security and network configurations. Push-based deployments to Kubernetes will also require a constant polling mechanism to check for health and status. But, one major advantage to push-based methods for Kubernetes is the ability for one central engine to interact with multiple clusters. This allows for easier admission control and data correlation across multiple clusters.

If a company wants to understand whether a push or pull-based deployment method is better for their application, they need to understand the benefits of each option, . The most important pieces to consider are the network and security requirements for the company. These two areas will often dictate which method will be the easiest to adopt.

Summary

This chapter finished the overview related to software delivery, deployment, and the importance of continuous processes. New concepts were also introduced related to GitOps, how GitOps is currently practiced in the industry, the difference between declarative executions and declarative state, and a high-level overview of push-based and pull-based deployments.

The goal for this chapter was to provide enough information for you to understand what GitOps is, common practices for GitOps, and some of the nuances in implementing GitOps that a team should consider as they begin their adoption of the GitOps processes.

Section 2: GitOps Types, Benefits, and Drawbacks

In the previous section, the fundamentals were covered and the foundation was set. Now, in this section, we will learn what different GitOps practices exist, how to choose the right one, how to define best practices, and where GitOps fits in.

This section comrpises the following chapters:

- *Chapter 4, The Original GitOps – Continuous Deployment in Kubernetes*
- *Chapter 5, The Purist GitOps – Continuous Deployment Everywhere*
- *Chapter 6, Verified GitOps – Continuous Delivery Declaratively Defined*
- *Chapter 7, Best Practices for Delivery, Deployment, and GitOps*

In this chapter, we're going to cover the following main topics:

- Original GitOps basics
- Kubernetes and operators
- Manifest explosion
- Benefits and drawbacks of originalist GitOps
- Common originalist GitOps tools

Original GitOps basics

The DevOps team was now in a mad scramble to get a GitOps process implemented and adopted. The development teams had continued to break down the old monolithic application into containers to deploy into a Kubernetes cluster, resulting in significant growth of deployable services across all clusters and teams. The haphazard service-oriented architecture style of deployments with tightly coupled version dependencies would soon result in a deployment process that is too heavy to support. This would require significant team growth and manual intervention just to achieve the business objectives.

By implementing Argo CD in the cluster, the team was able to quickly have deployments automatically sync with the cluster tied to the tool. Argo CD is a GitOps tool and closely aligns itself with the group that first coined the term GitOps in 2017. The original intent is to allow an in-cluster engine to execute the desired deployment, known as auto-syncing, with the Kubernetes cluster that the engine is tied to. The two main areas that the original GitOps method is most associated with are Kubernetes manifests, like `Helm`*, and Infrastructure-as-Code files, like Terraform.*

When there is a change in the code repository that houses either the Kubernetes manifest or the Terraform files, ArgoCD will auto-sync the changes with the cluster.

Since the team currently leverage Terraform and Kubernetes, the originalist style of GitOps would offer a quick solution for the DevOps team. The only thing that the team needed to better understand was how ArgoCD worked in Kubernetes in order to not be locked into that tool in the future.

Every Kubernetes cluster has a set of system components that help with cluster maintenance and administration. etcd and kube-api-server are some of these core components, but the components that actually execute most of the work in the cluster are the controllers.

Each controller in Kubernetes manages one or more resource types in the cluster. The management process consists of the controller reading what etcd says about that resource type, reading what the cluster shows relating to the resource type, and then working to make the two stay in sync with each other. Controllers will accomplish their tasks either by executing kube-api-server requests, or taking effects directly on the cluster. The complexity of the inner workings of Kubernetes are often uncovered the hard way, which is to say that they are uncovered during troubleshooting an issue. Therefore, it is highly recommended that anyone pursuing originalist GitOps practices should have a thorough working knowledge of Kubernetes first.

A popular originalist GitOps tool in the market is Argo CD, which leverages the operator model inside of the Kubernetes cluster. The goal of Argo CD is to make sure that the current state of a designated resource set inside of the cluster matches the desired end state that is declaratively defined in the Kubernetes manifests in a Git repository. This operator model with Argo CD is exactly the same as the controller model, but there is one significant difference: A controller treats etcd as the source of truth, while Argo CD treats the Git repository as the source of truth. The underlying behavior of Argo CD in relation to the Kubernetes cluster is essential to understand, especially when considering the administrative requirement for Kubernetes manifests and failure strategies.

By treating the git repository and `helm` *chart as the source of truth for the desired end state, the user should always be able to see what microservice configuration is in the cluster based on what the git repository shows. Also, since git has native versioning, a rollback to a previous version is as easy as reverting a version in git and letting Argo CD handle the deployment part.*

One other benefit that the team found was that if a cluster went down, since the `helm` *charts existed as the source of truth in the git repository, they could spin up another cluster and return it to the desired state in very little time. This shorter mean-time-to-restore for cluster recovery was a massive benefit for the company.*

1. *There were some drawbacks that the team found as well. They had a set of* `helm` *charts already created by the development teams, but the DevOps team didn't want to place the manifest creation and maintenance burden on the developers. Since Argo CD required that a chart existed in a single repo for every set of microservice resources that needed to be deployed to Kubernetes, there would be a manifest explosion.*

2. *Another drawback, which is significant for the engineering organization, is that the developers have a different cluster for every environment, meaning that there are four different clusters that need to be deployed to. Unfortunately, Argo CD does not easily allow for cross-cluster deployments of the same* `helm` *chart in sequential order. Adding to this issue, the company will require multiple production clusters, one for each major region of users across the globe. Argo CD does not have any way of being able to deploy the same chart across multiple clusters in different regions at the same time. The best option that the team could find was to leverage an Argo CD architecture known as an "app of apps," where a single git repository would store the files required to dictate applications and cluster, which requires manual executions only.*

3. *The last drawback, and probably the most important drawback the team found, was that the originalist GitOps process is only for deployments and specifically for the easiest part of the process. Every microservice manifest had to manually be created and tested with every values file combination to every environment and cluster in the architecture. Then, when the files were finalized and added to the git repository, Argo CD could be connected to execute the install command for the microservice. But all of the other required steps, such as approvals, tickets, testing, verification, and others, would have to be done outside of the GitOps process.*

Argo CD is a great GitOps solution and can solve many problems for different companies and teams in the industry. But the DevOps team needed to see if maybe there was a better GitOps tool that was native to Kubernetes and could handle all of their delivery and deployment requirements.

Jsonnet templatization examples


```
    "message": "My car is a Honda Civic"
  },
  "car2": {
    "make": "Toyota",
    "model": "Corolla",
    "message": "My car is a Toyota Corolla"
  }
}
```

Examples 5.5 and *5.6* show a basic set of files, the first being the Jsonnet file and the second being the output JSON file. The Jsonnet file starts with a function named `Car` that has two parameters named `make` and `model`. The function allows a user to specify the two parameters when calling the function, which is shown in the `car1: Car('Honda','Civic')` line. The output of the Jsonnet templating process creates a JSON file with the desired outcome of three car objects.

To allow a scalable set of JSON files, Jsonnet has the ability to use file inheritance, similar to modules or libraries for code.

Example 5.7: Jsonnet imports file:

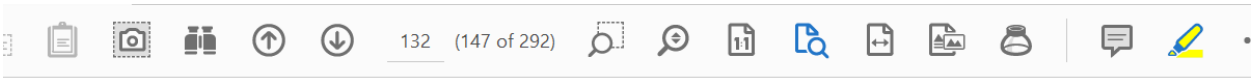
```
local superhero = import 'superhero.libsonnet';

{
  'Batman': superhero['Batman'],
  'Spiderman': {
    attributes: [
      { outfit: 'suit', color: 'blue and red' },
      { vehicle: 'motorcycle', color: 'blue and red' }
    ],
    sidekick: 'no',
    city: 'queens'
  }
}
```

Example 5.8: Jsonnet libsonnet file:

```
// SUPERHERO.LIBSONNET
{
  'Batman': {
```

engineer would fill out information in the libsonnet file. Then the administrator would statically define the required stage in the Jsonnet file and reference the predefined stage in the libsonnet file. This will output the desired output JSON file that the pipeline would then use. Because of the use of the libsonnet file, the administrator can allow significant configurability for the engineer while also enforcing a standard path that must be followed.

The execution engine that consumes the output JSON file would be able to leverage the declarative nature of the JSON file to provide repeatability and reliability. The Jsonnet templating capabilities add scalability to the entirety of the process, especially with regard to GitOps. The libsonnet file update can trigger the templating process, with the output JSON file being automatically uploaded to a predefined Git repository. Then, when the output JSON file is uploaded to a Git repository, that can trigger the desired pipeline to deploy.



Summary

This chapter provided a deeper dive into some of the basics of building a declarative file. Templating was also introduced to show how to add scalability to the benefits of a declarative approach to delivery and deployments. Templating is accomplished through templating engines, which are directed at specific declarative languages, namely Jsonnet for JSON and Helm for YAML. You should now know the difference between a flat file and a nested file, an object and an array, and XML, JSON, and YAML file design and syntax.

The next chapter will go through the basics of originalist GitOps with Visual Studio Code, Minikube, ArgoCD, GitHub Actions, and Flux.

App of Apps in ArgpCD and

Those are the basic requirements to get Argo CD set up inside of a cluster. For future Helm charts or Kubernetes manifests, steps in *Creating an application from a Git repository* and *Pushing a change* will need to be repeated for every application and manifest set.

Other than adding more manifest sets, the only other piece to consider is when it is needed to deploy to multiple clusters. One option is to install a different Argo CD set in each cluster and then leverage different logins for each cluster. That way can be rather tedious, but it would allow the same chart to be deployed to multiple clusters in parallel, if that was a desire. The alternative is to use a combination of an App of Apps deployment pattern and leveraging remote connection to each cluster's master node. An App of Apps is just that, an application in Argo CD that deploys other applications. The nesting of applications is a common way to overcome the lack of native multi-cluster and multi-environment support in Argo CD. The tutorial at `https://argoproj.github.io/argo-cd/operator-manual/declarative-setup/#app-of-apps` walks through this concept more.

Summary

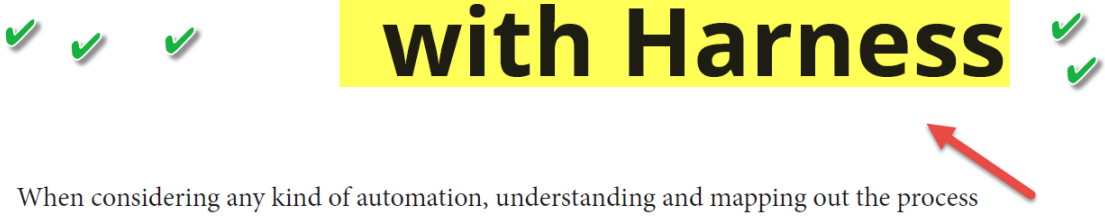
A tool such as Argo CD has a narrow support scope, being limited only to Kubernetes. But this narrow scope allows Argo CD to offer a quicker time-to-value, from installing to deploying. The capabilities of the tool have rightly earned it a reputation as one of the elite open source originalist GitOps tools widely used today.

The next chapter will cover how to implement verified GitOps. Integrating with multiple clouds and platforms, supporting an array of application architectures, and enforcing security and compliance are essential in today's technology-forward world. Verified GitOps aims to support these requirements in a declarative and automated way.

Finally a book/chapter on Harness !

10

Verified GitOps Setup – Continuous Delivery GitOps with Harness



When considering any kind of automation, understanding and mapping out the process is a hard requirement. Automation applied to an undefined process will result in gaps or holes that require manual intervention. Adopting a GitOps practice should start with documenting where automation is desired, how to define it in code, and how to trigger the execution.

The previous chapter walked through setting up originalist GitOps using Minikube, Helm, and ArgoCD. This chapter will look to set up verified GitOps, starting out with how to map the process and how to use declarative language for repeatability and reliability. Additionally, since an open source tool was used in the last chapter to show originalist GitOps, this chapter will use a vendor-based tool for verified GitOps.

Those are the basic requirements to get Argo CD set up inside of a cluster. For future Helm charts or Kubernetes manifests, steps in *Creating an application from a Git repository* and *Pushing a change* will need to be repeated for every application and manifest set.

Other than adding more manifest sets, the only other piece to consider is when it is needed to deploy to multiple clusters. One option is to install a different Argo CD set in each cluster and then leverage different logins for each cluster. That way can be rather tedious, but it would allow the same chart to be deployed to multiple clusters in parallel, if that was a desire. The alternative is to use a combination of an App of Apps deployment pattern and leveraging remote connection to each cluster's master node. An App of Apps is just that, an application in Argo CD that deploys other applications. The nesting of applications is a common way to overcome the lack of native multi-cluster and multi-environment support in Argo CD. The tutorial at `https://argoproj.github.io/argo-cd/operator-manual/declarative-setup/#app-of-apps` walks through this concept more.

Summary

A tool such as Argo CD has a narrow support scope, being limited only to Kubernetes. But this narrow scope allows Argo CD to offer a quicker time-to-value, from installing to deploying. The capabilities of the tool have rightly earned it a reputation as one of the elite open source originalist GitOps tools widely used today.

The next chapter will cover how to implement verified GitOps. Integrating with multiple clouds and platforms, supporting an array of application architectures, and enforcing security and compliance are essential in today's technology-forward world. Verified GitOps aims to support these requirements in a declarative and automated way.

In this chapter, we're going to cover the following main topics:

1. Mapping out the process
2. One manifest or many
3. A manifest for integrations
4. A manifest for configuration
5. A manifest for execution
6. A manifest for delivery
7. Verified GitOps with Harness

Mappings!

The team section of the map can be represented as a person icon with an appropriate title underneath:

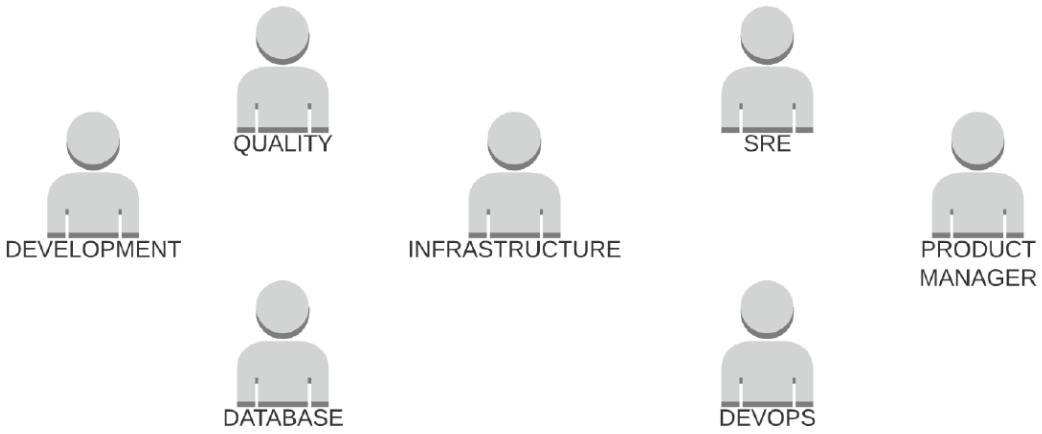


Figure 10.2 – Teams involved in the process

The tools section will be similar to the team section:

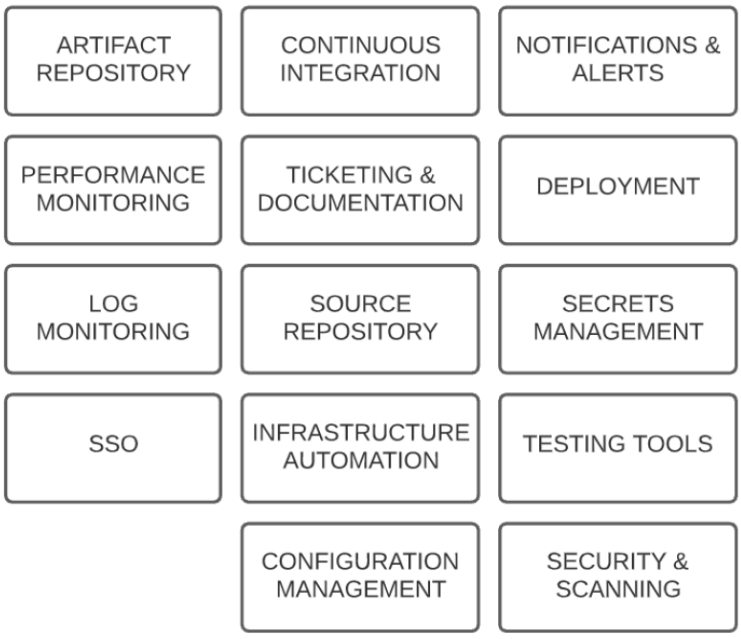


Figure 10.3 – Process steps

After mapping the high-level requirements between the platforms, tools, and teams, the next step is to highlight which areas are not automated currently:

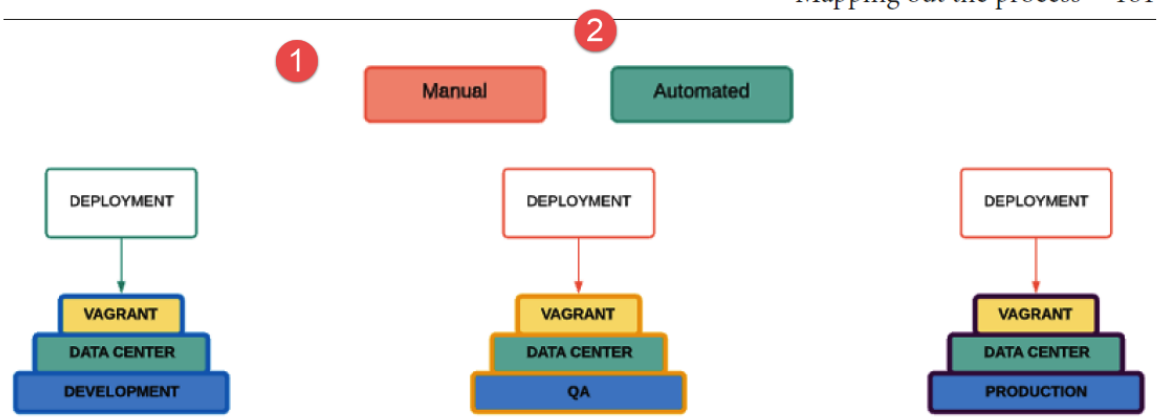


Figure 10.4 – Manual or automatic deployments

Then, to understand the current state of the delivery, every step of the delivery process will need to be tagged with either a manual, automatic, or partial execution type. This will help understand what can be or needs to be automated later:

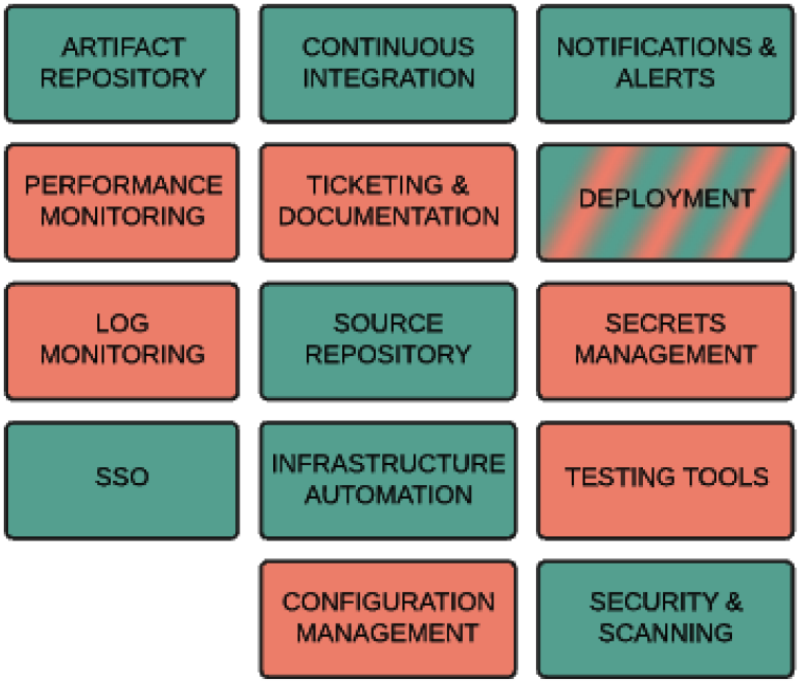


Figure 10.5 – Manual or automatic steps

3. **Pipeline process map**: With the tools, platforms, and teams defined, the last step in process mapping is putting together the pipeline process for the delivery and deployment:

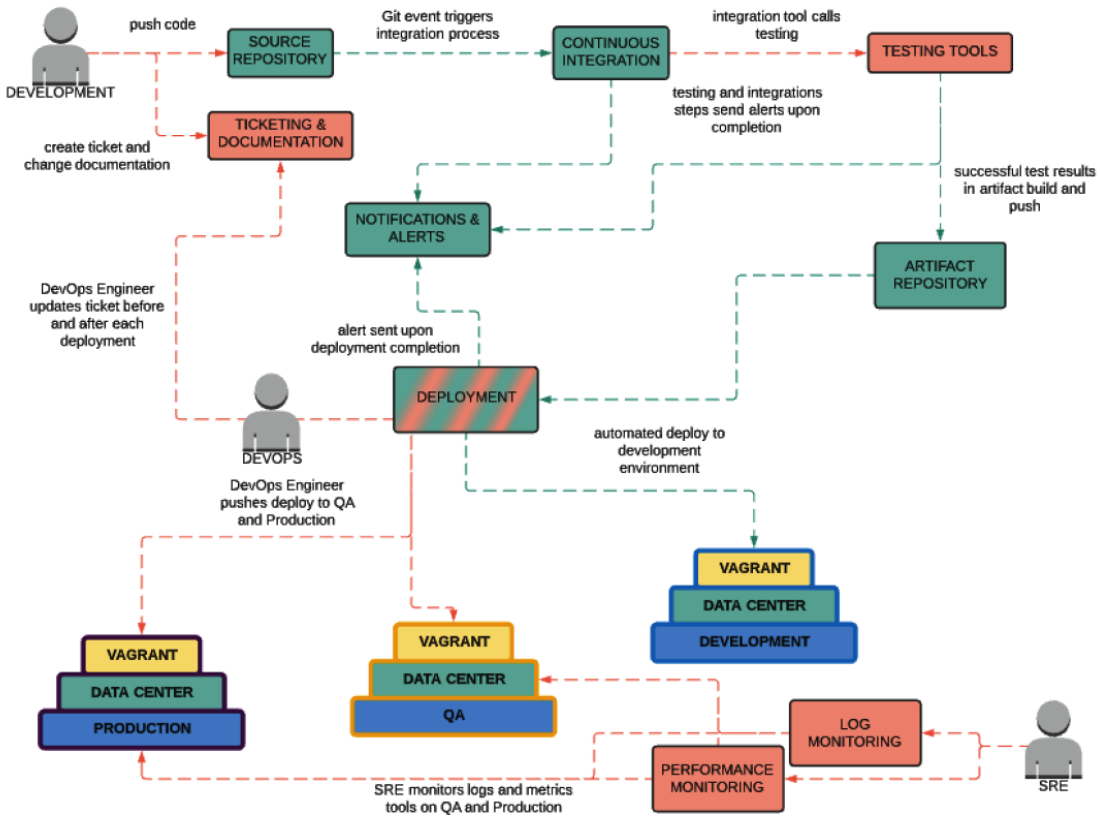


Figure 10.6 – Process map showing steps, teams, environments, and current execution types

Most companies fit into one of two categories:

1. Do what you want, but be successful.
2. Do what we want, we are successful.

1 The first company type desires to give complete freedom of technology decisions to the developer teams. The teams can choose which platforms they use, which tools they use, how they operate and administer the technology, how they monitor the application, and even which programming language they build the application with. The trade-off for this level of freedom is that the team has to be successful based on pre-defined business metrics. There are some teams that thrive in a **wild west** style of DevOps, but when all administration, management, and support is delegated to the same team that should also be developing the application, the desire to have a central support group becomes ever more important.

2 The second company type prefers to have the developers focus on getting the application into the hands of users, without worrying about what tools or platform is required. The company will define the technology stack, hire only those with specified programming knowledge and experience, and employ a centralized DevOps team that focuses on the developers being unimpeded in their work. The drawback to a more opinionated approach is that developers have little freedom to innovate with their application functionality and feature set.

Every engineering organization will have a different approach to implementing verified GitOps, depending on how they choose to account for the different areas of balance mentioned in this chapter. But one of the opinionated parts of Harness is that the manifest size and number is predetermined for the company in many ways. This means that the way that multiple manifests are used to accomplish a desired outcome will be the same everywhere. Although some engineering organizations might want to implement a smaller-number-of-large-files approach to verified GitOps, it is important to see what verified GitOps in Harness can look like.

Verified GitOps with Harness

The DevOps team was almost finished with building out their Kubernetes-based applications in Harness. To do this, the team had to add the artifacts for their microservices as Harness Services, which combined their core Docker artifact, the Helm chart, and any overrides. After the Services were created, the team then added the different clusters they needed to deploy to, as well as the designated namespaces. Then they moved on to creating the different workflows that were required for deployments across the different environments. These workflows had to include any business, security, or compliance requirements for the appropriate environments. The DevOps team had to also include any steps required for pre-deployments and post-deployments from the developers, the quality engineers, and the cloud infrastructure team. After these were completed, the team created a full delivery pipeline that would deploy all of the workflows in sequential order, without any intervention by a user until the production deployment. The last thing to add was the trigger for the pipeline to execute.

A major benefit of using a tool like Harness is the built-in capabilities around notifications, failure strategies, and some of the verification benefits. The DevOps team was trying to figure out how to get an automated rollback working in Ansible, which would be triggered on either a timing limit or an error message. They also needed to consider the verification process after the production deployment, which the business mandated for the delivery process. The DevOps team had figured that they would work on the automation for that at a later time and instead have significant alerts and continually refreshed dashboards on their monitors to watch the production environment. Harness has an automated verification process out of the box, which made for an easy win for the team. The last piece, which was the easiest to solve, was the ability for different execution events to notify the appropriate teams and channels. Ansible, and almost any tool for that matter, has this ability. Harness ties well into their chat tool, can email their teams as needed, and can also send alerts to their incident management solution as well.

After achieving continuous delivery with their Kubernetes applications, the DevOps team would move on to the serverless and then the traditional applications next. With the triggering of the delivery pipelines coming from the Git repository event, the developers would rarely, if ever, access Harness or even press a deployment button. With all of the validation, reporting, standardization, and automation that Harness provides the company, they would have minimal administration time requirements moving forward.

Harness is structured in a multi-layer abstraction model, where basic building blocks are added and then combined to make larger objects. The structure of Harness looks similar to a pyramid.

Everything in Harness is based on a large number of native integrations, platforms supported, and templates. Those integrations, platforms, and templates can then be combined to make up the Services and Environments in Harness. The Workflows combine the two layers below it, as well as workflow-specific capabilities, into a set of pre-deployment, deployment, and post-deployment steps. Lastly, the Pipelines layer combines workflows, with optional approval steps, to create the final layer of abstraction.

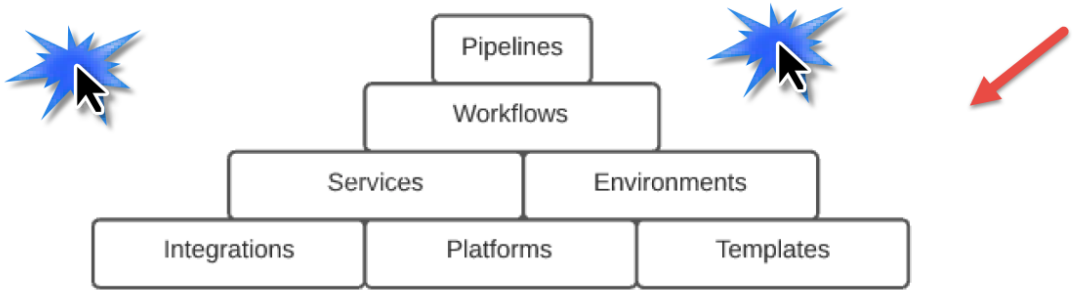


Figure 10.18 – Harness entity relationship

After signing up for a Harness account, one of the first steps will be to add an execution agent, known as a Delegate, to a location inside of the company network or **virtual private cloud** (**VPC**). The Delegate can be installed in a Kubernetes cluster, an ECS cluster, or on a Linux server:

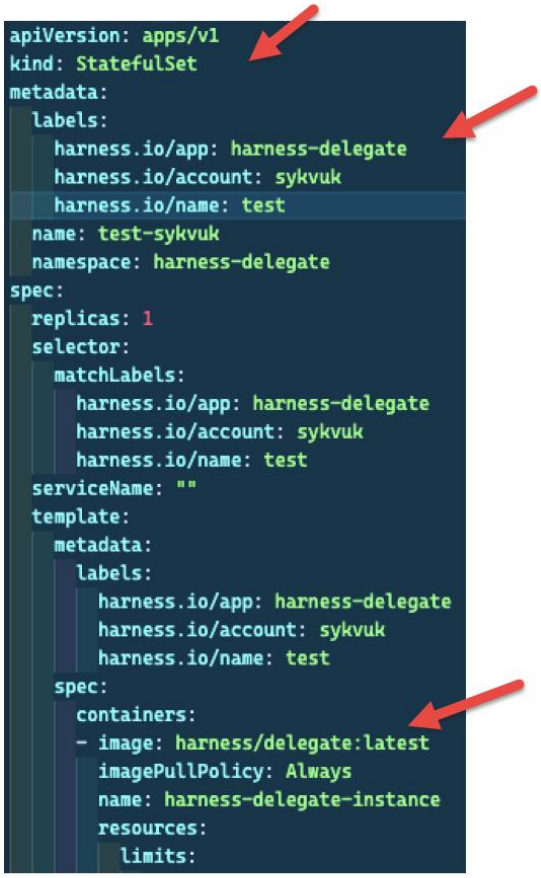


Figure 10.19 – Delegate YAML

To install the delegate into the Minikube cluster, download the YAML from Harness and open the file in Visual Studio Code. Then you will want to change the memory resource requirements to 4 Gi:

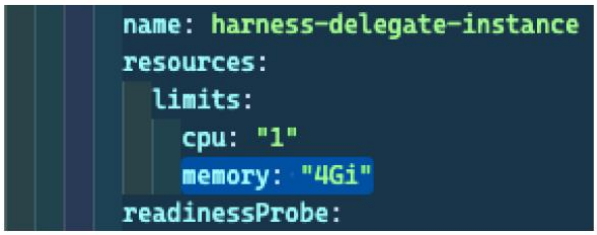


Figure 10.20 – Memory resource update

integrations

Once that change is done and the file is saved, while running `kubectl`, applying `-f harness-delegate.yaml` will install the Delegate into the cluster:

```
(base) ➜  ~ kubectl apply -f ~/Downloads/harness-delegate-kubernetes/harness-delegate.yaml
namespace/harness-delegate created
clusterrolebinding.rbac.authorization.k8s.io/harness-delegate-cluster-admin configured
secret/bf-test-proxy created
statefulset.apps/bf-test-sykvuk created
```

Figure 10.21 – Deploying the Delegate into Minikube

The Delegate will then execute all instructions that Harness gives it:

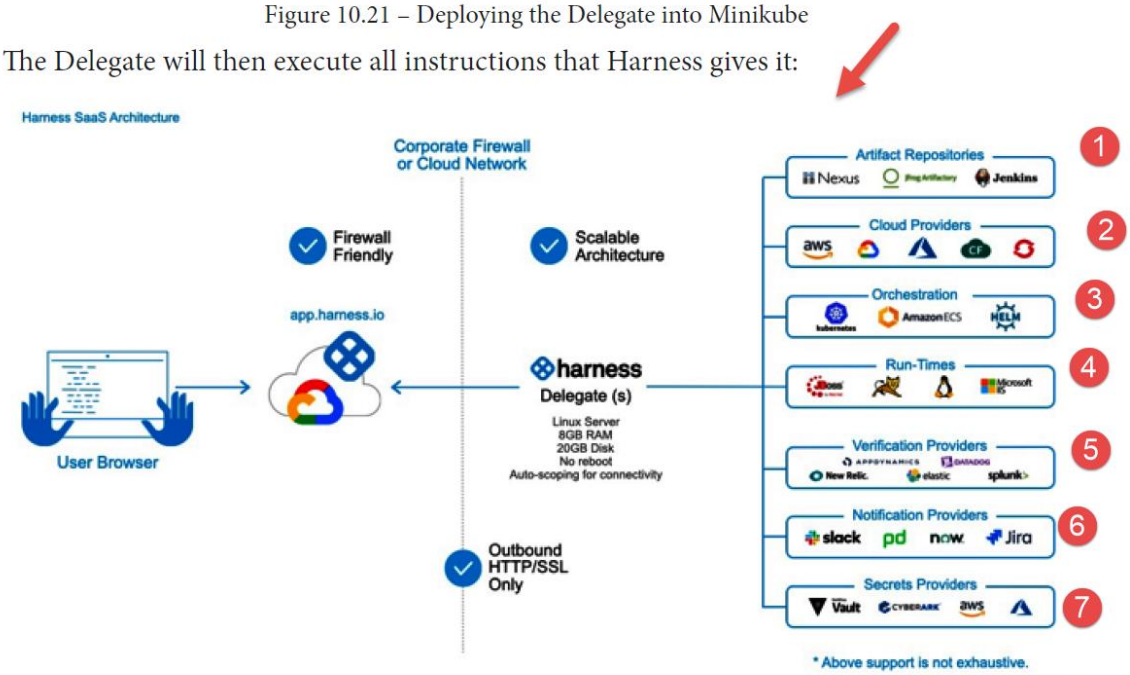


Figure 10.22 – Delegate connection

After the Delegate is installed, the next step is to connect to the required integrations and platforms using the appropriate connection credentials and permissions set. The first of these integrations is the Kubernetes cluster:

After anything in Harness is created, there is an automatically generated configuration YAML file. These configuration files can be accessed in Harness or by having Harness push the files to the designated Git repository:

1. **Configuration As Code in Harness**: The configuration code files in Harness are at the top right of the account setup page:

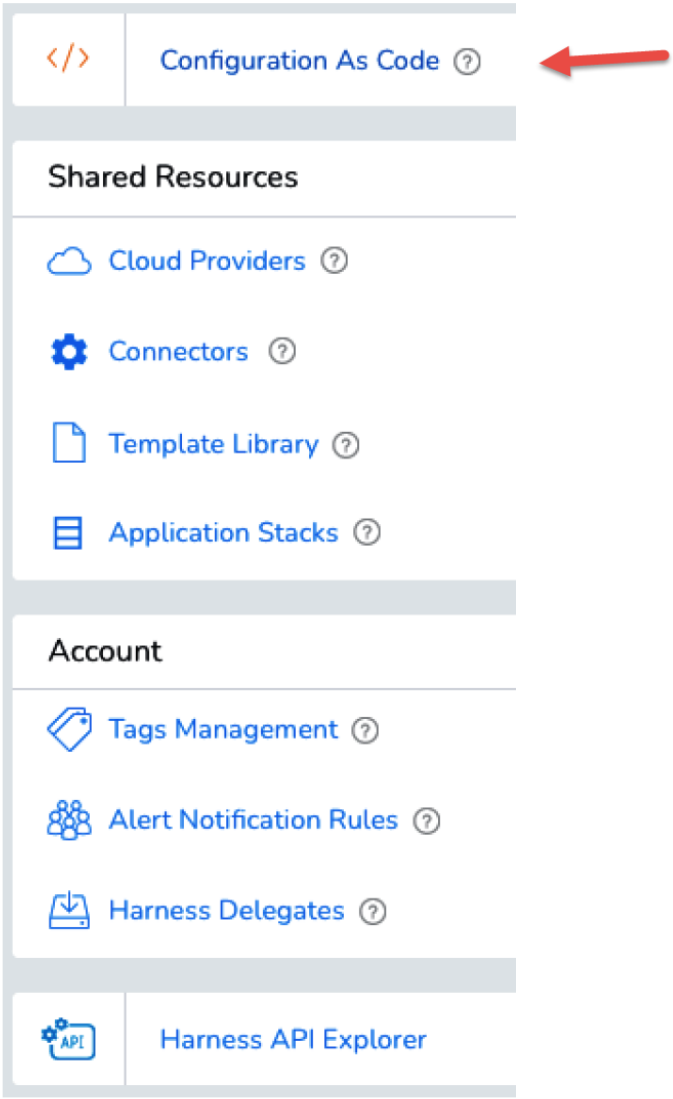


Figure 10.42 – Configuration As Code on the Harness setup page

In the Configuration As Code section of Harness, any file can be selected to show the code representation of that object, such as a Harness Service:

The structure of files is similar in the Git repository, with the `Application` folder living in the **Setup/Applications** structure. Inside the `Application` folder are all objects belonging to the application as well. Other configuration code files that exist are the artifact repositories, the Git repositories, the verification providers, platforms, and so on. With every integration, platform, and every other object in Harness being automatically converted into declarative code, the ability to achieve a verified GitOps practice is easily attained.

When considering everything from this chapter, there are some important items to consider between Ansible and Harness, as well as best practices when adopting a verified GitOps practice.

Ansible Pros and Cons

Ansible:

1. **Pros**: Ansible is an open source solution, meaning that it doesn't require licenses to be purchased. It also has a major benefit of being almost infinitely configurable, since it can run scripts. Any desired integration can be added to Ansible, as long as there is an API or CLI. In fact, many of these integrations are already built out in the plugins.
2. **Cons**: Although Ansible is very extensible, there are some important drawbacks to the tool. The first of the drawbacks, and probably the most important, is that Ansible has a very expensive setup, configuration, and administration cost. Because everything after the initial install is manually configured, it can consume many hours for multiple engineers to build out a delivery pipeline that can deploy to multiple different endpoints. As with any open source solution, the users must host the whole solution, which means that there are hosting and scaling costs. Lastly, even though Ansible does not have a licensing fee, there are significant support costs if community support is not sufficient.

11

Pitfall Examples – Experiencing Issues with GitOps

A major section of this book was dedicated to covering the high-level understanding of different GitOps practices, how they are implemented, as well as benefits and drawbacks. This chapter is intended to show some of the drawbacks of GitOps in a more practical way.

To follow along with this chapter, you will need to follow the steps in *Chapter 9*, *Originalist Gitops in Practice – Continuous Deployment*, related to getting minikube, a Git repository, and Argo CD set up and running. The assumption throughout this chapter is that the user has Argo CD running in the minikube cluster and Argo CD connected to a Git repository. Although Argo CD is the main tool used for this chapter, that is only because Argo CD is the lightest and quickest to get set up and show pitfalls. This chapter is not intended to show issues with Argo CD, but rather pitfalls that can accompany GitOps in general.

- **SELF HEAL**: Checked – If a change of a workload is enacted on the cluster side, rather than on the Git repository side, Argo CD will correct the workload in the cluster to match what is in the Git repository:

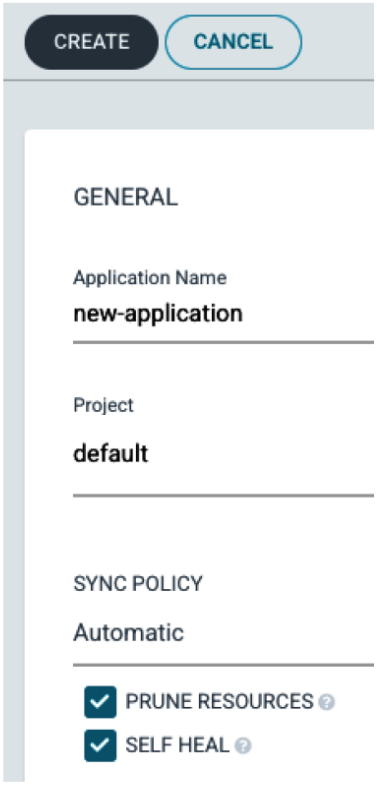


Figure 11.1 – Argo CD application with automatic sync, prune resources, and self-heal options

With those settings configured, the application will be deployed successfully:

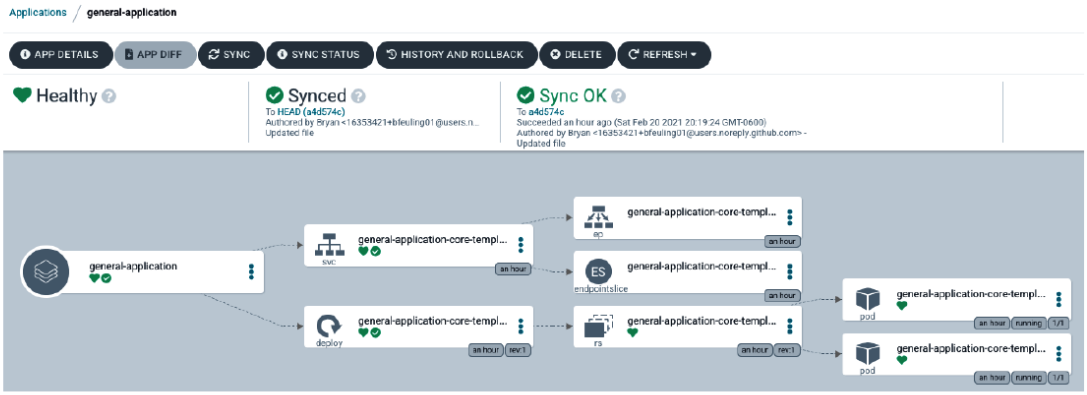


Figure 11.2 – Healthy application resources

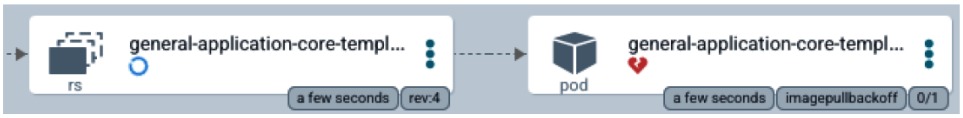


Figure 11.26 – Recreated replica set with a bad replica

Revert the changes to make the Git repository and cluster match, allowing Argo CD to be put back into a successful state:

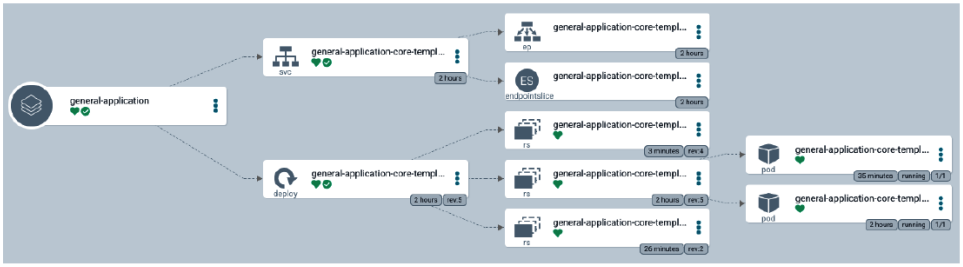


Figure 11.27 – Healthy application in Argo CD

Notice that there are now two abandoned replica sets in the cluster.

Building out Helm Charts is not always an easy feat to accomplish, often requiring multiple iterations to get the Charts correct. If there is a desire to move toward an originalist GitOps practice, beware of this type of pitfall, which can tie directly into automated failure strategies.

Failure strategies

A major requirement for the DevOps team is to have a way to trigger a rollback automatically when any issue occurs in the production environment. The company has a strict service-level agreement that specifies that their Software as a Service product has 99.99% uptime. Therefore, if their application is unavailable for more than 5 minutes a month, that would break the uptime requirement.

The team has researched the most common issues that would break a production deployment, how to test for them, and what they need to monitor to account for any other issues that might show up. The problem they have is how they would trigger a rollback in a GitOps model automatically.

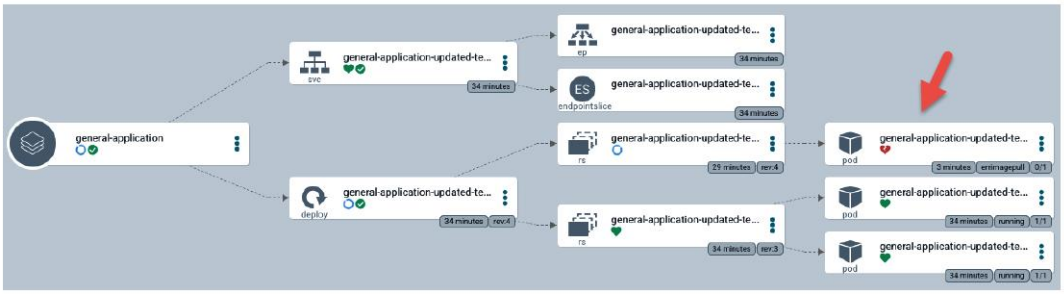


Figure 11.33 – Argo CD showing a broken rollout

Even though the rollback was executed within the cluster, Argo CD re-enforced the broken version in the Git repository. To overcome this issue, the "self-heal" option needs to be turned off.

3. **Turn off self-healing** – In Argo CD, go to the application's app details, edit the details, and turn off self-healing:

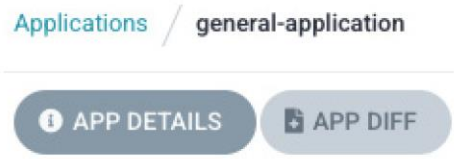


Figure 11.34 – Argo CD application details

Clicking **EDIT** will give the editable configurations for the application:

Figure 11.35 – Editing the application details

Different application details can be changed, such as disabling self-healing:

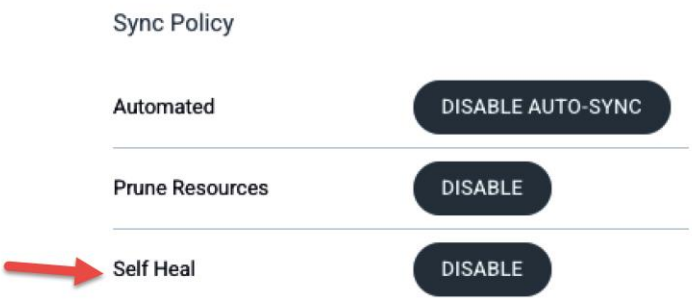


Figure 11.36 – Application automation features

With the update in the Git repository matching what is in the cluster, Argo CD will now show that the application is in sync:

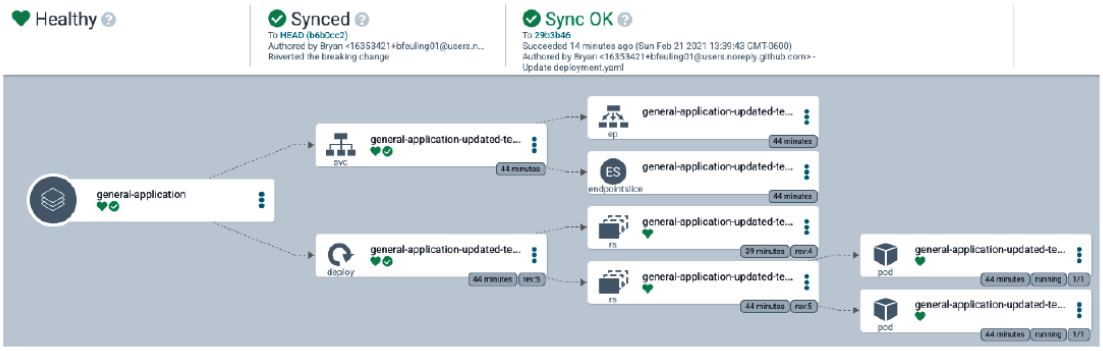


Figure 11.43 – Healthy application in Argo CD

This failure strategy limitation is consistent across all originalist GitOps tools, since the tools treat the Git repository as the source of truth and have no way of executing an automated rollback with an accompanying update to the Git repository to match the outcome of the rollback. As such, if there is a desire for automated rollbacks, then either the self-healing feature of the GitOps tool needs to be disabled and the rollback command must be executed upon an issue, or there needs to be a process of alerting when there is an issue, and a user reverts the change as needed.

If the rollback should be triggered automatically within the cluster, then a testing process needs to run, be evaluated, and then roll back when there is an issue. Another issue with releases is authorizing deployments into higher-level environments, such as through an approval or ticketing process.

Governance and approvals

As the DevOps team goes through the process mapping phase, one of the most important requirements that the company and the security team have is support for advanced governance and auditing throughout the delivery process. Because of the different data protection and privacy laws across the world, as well as financial auditing requirements, there are restrictions associated with user access, deployment approvals, interaction auditing, and execution auditing.

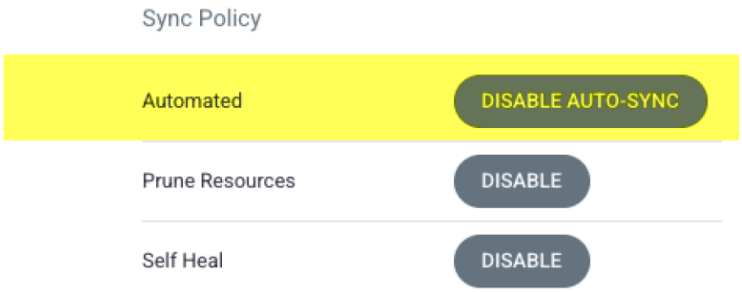


Figure 11.45 – Disabling auto-sync in the application details

2. **Approval through a pull request and auto-sync** – If auditing both who has access to the clusters and what actions are taken on the cluster is required, then leveraging the auto-sync and self-heal options of Argo CD will prevent users from making any permanent changes in the cluster. If the audit has an approval requirement for deployment, then originalist GitOps tools will have no native way to meet this requirement. Instead, the company could leverage the approval of a Git repository pull request.

   Create a new branch, which is a Git-native way of creating a clone of the code to work with while avoiding any changes from affecting the main branch, in the Git repository by typing in the desired name of the new branch and selecting the **Create branch** option:

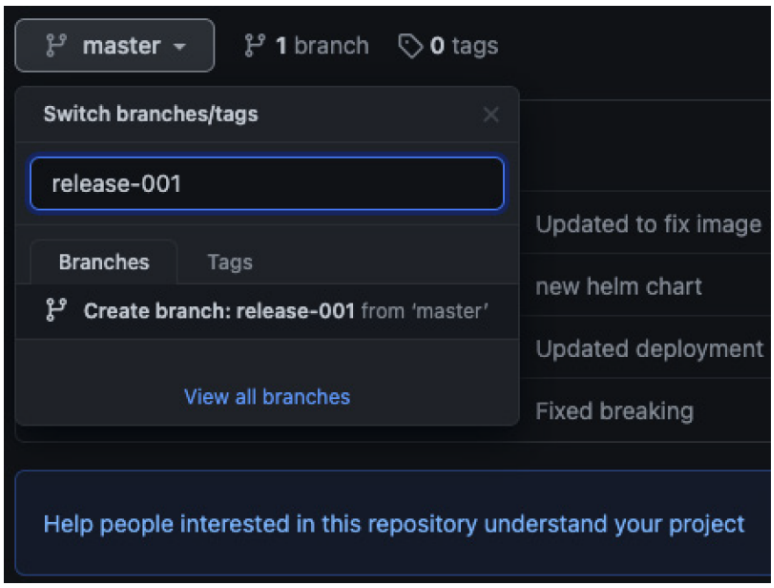


Figure 11.46 – Creating a new branch

```yaml
---
harnessApiVersion: '1.0'
type: DOCKER
imageName: library/busybox
serverName: docker

---
harnessApiVersion: '1.0'
type: APPLICATION_MANIFEST
gitFileConfig:
  branch: master
  connectorName: helm-charts
  useBranch: true
  useInlineServiceDefinition: false
storeType: Remote

---
harnessApiVersion: '1.0'
type: INFRA_DEFINITION
cloudProviderType: KUBERNETES_CLUSTER
deploymentType: KUBERNETES
infrastructure:
- type: DIRECT_KUBERNETES
  cloudProviderName: default-cluster
  namespace: default
  releaseName: release-${infra.kubernetes.infraId}
```

Figure 11.56 – Harness Service YAML

Once the different core requirements are configured in Harness, then the deployment definition can be created, which links all of the previous core requirements into a longer YAML file:

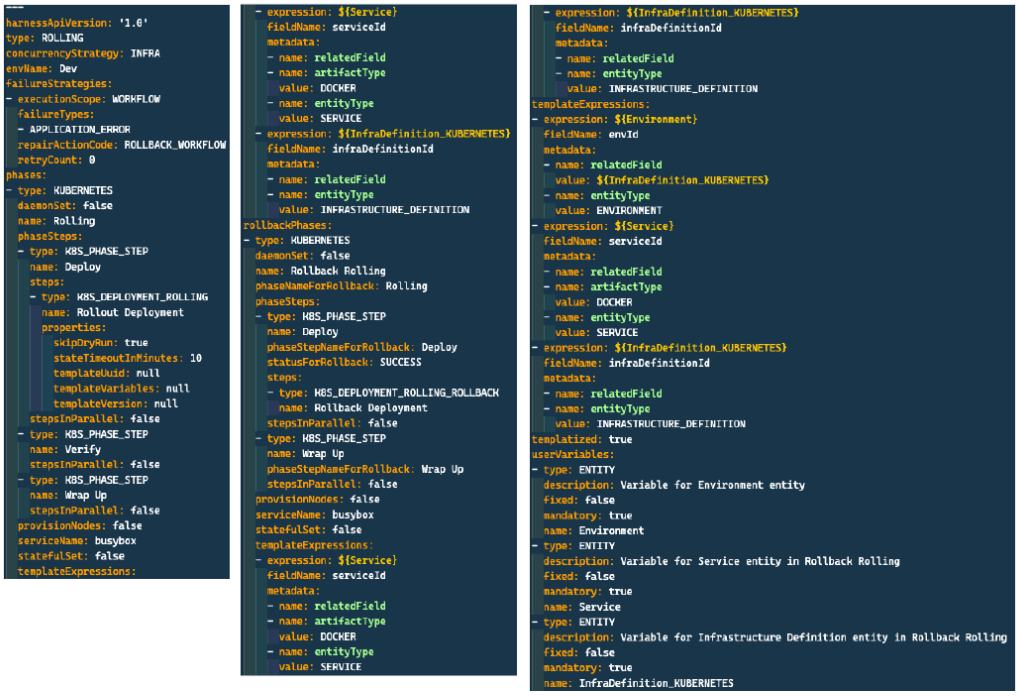


Figure 11.57 – Harness workflow YAML

Although the scope for both Ansible and Harness is significantly larger when compared to Argo CD and other originalist GitOps tools, the accompanying configuration requirements can also be extensive. Ansible and Harness have ways of mitigating these configuration requirements, whether that is auto-generation of the files, such as in Harness, or extensive templating capabilities.

Summary

By understanding the pitfalls that can occur, regardless of which GitOps practice is implemented, the team can better consider what will need to be done to avoid issues. Another major benefit of understanding these pitfalls is the ability to accurately understand how a tool or practice achieves the desired outcome, and what the required effort will be.

The next chapter will be a summary chapter that talks through what the book has covered, lessons learned throughout the book, and what the next steps look like for those wanting to implement GitOps in their company and organization.

Continuous deployment GitOps – originalist and purist

One consideration that the DevOps team had was the possibility of tying in a tool like Argo CD with Ansible or Harness. This integration would allow the teams using Kubernetes to still have the continuous delivery portion automated, but also allow for the self-healing and automated pruning features of Argo CD to enforce the desired end state on the clusters during and after a deployment. But exploring this integration resulted in two main concerns related to inconsistent processes and automated execution led the DevOps team to abandon the idea. They realized that since Argo CD only works on Kubernetes clusters, they would have an inconsistent process with any application that was not Kubernetes-based. Also, Argo CD can only leverage the automated pruning and self-healing features if the automated syncing feature is turned on.

What the team wanted, but couldn't find, was a deployment tool like Argo CD, with the automated pruning and self-healing capabilities, but across all of the different application architecture. Terraform has some capability for this, by leveraging a state file, but the enforcement would only happen on the next execution of Terraform, rather than enforcing the desired configuration immediately. In reality, no other tool has the capability that Kubernetes-based GitOps tools have, such as Argo CD or Flux.

If the core of all GitOps practices is the use of declarative language files that are stored in a source code repository, then any tool that can accomplish or leverage these files becomes a de facto GitOps tool. However, certain tools that advertise themselves as GitOps tools, or are known in the industry as GitOps tools, are typically those tools that are native to Kubernetes, such as Argo CD and Flux. These tools have a reputation of providing low-touch operation, since they are focused solely on enforcing a desired outcome, and what they sacrifice to fulfill the low-touch operation capability is fewer capabilities and options for the end users. The most obvious capability limitation is that the tools only work on one platform. But another limitation is the ability to natively support pre-deployment and post-deployment requirements that a company might need, such as auditing, ticketing, testing, and so on. But, since GitOps tools such as Argo CD and Flux are focused solely on Kubernetes, they fit the most common definition of GitOps in the industry, which is the originalist GitOps practice. Originalist GitOps is a practice that focuses solely on continuous deployment, on a Kubernetes cluster, and leverages only Git repositories as the source of truth.

Continuous deployment GitOps – originalist and purist

One consideration that the DevOps team had was the possibility of tying in a tool like Argo CD with Ansible or Harness. This integration would allow the teams using Kubernetes to still have the continuous delivery portion automated, but also allow for the self-healing and automated pruning features of Argo CD to enforce the desired end state on the clusters during and after a deployment. But exploring this integration resulted in two main concerns related to inconsistent processes and automated execution led the DevOps team to abandon the idea. They realized that since Argo CD only works on Kubernetes clusters, they would have an inconsistent process with any application that was not Kubernetes-based. Also, Argo CD can only leverage the automated pruning and self-healing features if the automated syncing feature is turned on.

What the team wanted, but couldn't find, was a deployment tool like Argo CD, with the automated pruning and self-healing capabilities, but across all of the different application architecture. Terraform has some capability for this, by leveraging a state file, but the enforcement would only happen on the next execution of Terraform, rather than enforcing the desired configuration immediately. In reality, no other tool has the capability that Kubernetes-based GitOps tools have, such as Argo CD or Flux.

If the core of all GitOps practices is the use of declarative language files that are stored in a source code repository, then any tool that can accomplish or leverage these files becomes a de facto GitOps tool. However, certain tools that advertise themselves as GitOps tools, or are known in the industry as GitOps tools, are typically those tools that are native to Kubernetes, such as Argo CD and Flux. These tools have a reputation of providing low-touch operation, since they are focused solely on enforcing a desired outcome, and what they sacrifice to fulfill the low-touch operation capability is fewer capabilities and options for the end users. The most obvious capability limitation is that the tools only work on one platform. But another limitation is the ability to natively support pre-deployment and post-deployment requirements that a company might need, such as auditing, ticketing, testing, and so on. But, since GitOps tools such as Argo CD and Flux are focused solely on Kubernetes, they fit the most common definition of GitOps in the industry, which is the originalist GitOps practice. Originalist GitOps is a practice that focuses solely on continuous deployment, on a Kubernetes cluster, and leverages only Git repositories as the source of truth.

No perfect tool !

Harness:

1. **Pros**: Harness has a very light cost of ownership, both with regard to hosting and administration requirements. It also has significant customer support available and included through different chat options and a growing community. Lastly, the abundance of advanced capabilities and extensive integrations make for an easy-to-scale solution that provides confidence and reliability across all software delivery.
2. **Cons**: The drawbacks to using a tool like Harness are the common drawbacks associated with a vendor solution. The first, and probably most noticeable, is that there is a license fee to leverage the tool in a useful way. The other drawback is that the solution is closed source, which means that non-Harness engineers cannot alter the software code whenever they choose to meet their company's requirements. The only way to work around this issue is to either code the extensions themselves or to submit a feature request to the company.

There are no perfect tools, especially in the world of continuous delivery and GitOps. This is especially true when understanding that every tool will be implemented in a different way by every user and company. What is very important when considering a solution for continuous delivery and verified GitOps is to have an honest understanding of the benefits and drawbacks associated with the tools. Understanding both the pros and cons will aid in the proper analysis of which tool should be used and also what to look out for when implementing that tool.

Summary

This chapter discussed the different concepts related to building out verified GitOps using Ansible and Harness. Emphasis was placed on the different areas of balance that exist across manifest size and number, configurable and opinionated pipelines, and so on. Finally, the chapter went a bit deeper into how verified GitOps is shown in Harness.

After discussing originalist GitOps in depth in the last chapter, and verified GitOps in this chapter, it is important to show the pitfalls of GitOps. By understanding common issues that are encountered on the way to implementing GitOps, a team can properly navigate their GitOps implementation, which will be shown in the next chapter.

The structure of files is similar in the Git repository, with the `Application` folder living in the **Setup/Applications** structure. Inside the `Application` folder are all objects belonging to the application as well. Other configuration code files that exist are the artifact repositories, the Git repositories, the verification providers, platforms, and so on. With every integration, platform, and every other object in Harness being automatically converted into declarative code, the ability to achieve a verified GitOps practice is easily attained.

When considering everything from this chapter, there are some important items to consider between Ansible and Harness, as well as best practices when adopting a verified GitOps practice.

Ansible Pros and Cons

Ansible:

1. **Pros**: Ansible is an open source solution, meaning that it doesn't require licenses to be purchased. It also has a major benefit of being almost infinitely configurable, since it can run scripts. Any desired integration can be added to Ansible, as long as there is an API or CLI. In fact, many of these integrations are already built out in the plugins.
2. **Cons**: Although Ansible is very extensible, there are some important drawbacks to the tool. The first of the drawbacks, and probably the most important, is that Ansible has a very expensive setup, configuration, and administration cost. Because everything after the initial install is manually configured, it can consume many hours for multiple engineers to build out a delivery pipeline that can deploy to multiple different endpoints. As with any open source solution, the users must host the whole solution, which means that there are hosting and scaling costs. Lastly, even though Ansible does not have a licensing fee, there are significant support costs if community support is not sufficient.