

Continuous Deployment with Argo CD, Jenkins X, and Flux

GitOps and Kubernetes

Billy Yuen
Alexander Matyushentsev
Todd Ekenstam
Jesse Suen



MANNING



MEAP Edition
Manning Early Access Program
GitOps and Kubernetes
Continuous Deployment with Argo CD, Jenkins X, and Flux
Version 8

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux and participating in the Manning Early Access Program (MEAP)!

GitOps was first described in a series of blog posts by Alexis Richardson in 2017 and is a relatively new and actively emerging topic. Since then, GitOps has been getting an increasing amount of mindshare within the cloud-native community, with lots of articles, blogs, and conference presentations covering the topic. But it felt to us that none of them fully articulated all the issues, considerations, and best-practices required to implement GitOps at enterprise scale effectively.

This book aims to provide readers with a practical guide to implementing a GitOps-based system for managing and operating Kubernetes clusters. We cover fundamental GitOps concepts, such as Environment Management, Access Control & Security, Pipelines, and Observability. There are also chapters on advanced topics such as Deployment Strategies, Secrets, and Considerations for Enterprise Scale. At the end of the book, we tie it all together and provide a step-by-step tutorial for you to learn how to set up a full GitOps CI/CD pipeline for a service using three of the more popular GitOps tools: Argo CD, Jenkins X, and Flux.

To get the most benefit from this book, you'll want to have an understanding of current software development practices (CI/CD, etc.), Git revision control, and basic Unix shell scripting. It would also be beneficial to have some experience with Kubernetes and containers. Please let us know your thoughts about what we've written so far by posting in the [liveBook discussion forum](#). Your feedback is essential to ensuring GitOps and Kubernetes is the best book possible.

brief contents

PART I - BACKGROUND

- 1 *Why GitOps?*
- 2 *Kubernetes & GitOps*

PART II - PATTERNS & PROCESSES

- 3 *Environment Management*
- 4 *Pipelines*
- 5 *Deployment strategies*
- 6 *Access Control & Security*
- 7 *Secrets*
- 8 *Observability*

PART III - TOOLS

- 9 *Argo CD*
- 10 *Jenkins X*
- 11 *Flux*

APPENDICES

- A *Setup a Test Kubernetes Cluster*
- B *Setup GitOps Tools*
- C *Configure GPG Key*

1

Why GitOps?

This chapter covers:

- **What is GitOps?**
- **Why GitOps is important**
- **Comparison of GitOps with other approaches**
- **Benefits of GitOps**

Kubernetes is a massively popular open-source platform that orchestrates and automates operations. While it improves the management and scaling of infrastructure and applications, Kubernetes still has challenges managing the complexity of releasing applications frequently. Git is the most widely used version control system in the software industry today.

GitOps is a set of procedures that use the power of Git to provide both revision and change control within the Kubernetes platform. A GitOps strategy can play a big part in how quickly and easily teams manage their services' environment creation, promotion, and operation.

Using GitOps with Kubernetes is a natural fit, with the deployment of declarative Kubernetes manifest files being controlled by common Git operations. GitOps brings the core benefits of infrastructure-as-code and immutable infrastructure to the deployment, monitoring, and lifecycle management of Kubernetes applications in an intuitive and accessible way.

1.1 Evolution to GitOps

Two everyday tasks in managing and operating computer systems are infrastructure configuration and software deployment. Infrastructure configuration prepares computing resources (e.g., servers, storage, load balancers) for the software application to operate correctly. Software deployment is the process of taking a particular version of a software application and making it ready to run on the computing infrastructure. Managing these two

processes is at the core of what GitOps is. But before we dig into how this is done in GitOps, it is useful to understand the challenges that have led the industry towards DevOps and the immutable, declarative infrastructure of GitOps.

1.1.1 Traditional Ops

In a traditional information technology operations model, development teams are responsible for periodically delivering new versions of a software application to a QA team that tests the new version and then delivers to an operations team for deployment. New versions of software might be released once a year, once a quarter, or at some other shorter interval. It becomes increasingly difficult for a traditional operations model to support increasingly compressed release cycles.

The operations team is responsible for the infrastructure configuration and deployment of the new software application versions onto that infrastructure. The operations team's primary focus is to ensure the reliability, resilience, and security of the system running the software. Without sophisticated management frameworks, infrastructure management can be a difficult task that requires a lot of specialized knowledge.

IT OPERATIONS: The set of all processes and services that are both provisioned by an IT staff to their internal or external clients and used by themselves to provide a business's technology needs. Operations work can include responding to tickets generated for maintenance work or customer issues.¹

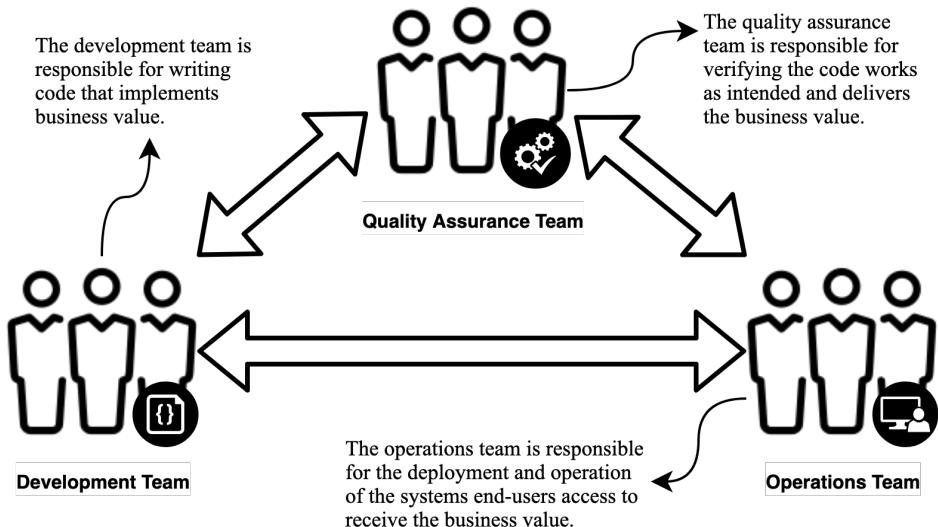


Figure 1.1 Traditional IT teams are typically composed of separate Development, Quality Assurance, and Operations teams. Each team specializes in a different aspect of the application development process.

¹https://en.wikipedia.org/wiki/Data_center_management#Operations

Because three teams are involved, often with different management reporting structures, a detailed hand-off process and thorough documentation of the application changes are needed to ensure the application is adequately tested, appropriate changes are made to infrastructure, and the application is installed correctly. However, this leads to deployments taking a long time and reduces the frequency deployments can be done. Also, with each transition between teams, the possibility of essential details not being communicated increases, possibly leading to gaps in testing or incorrect deployment.

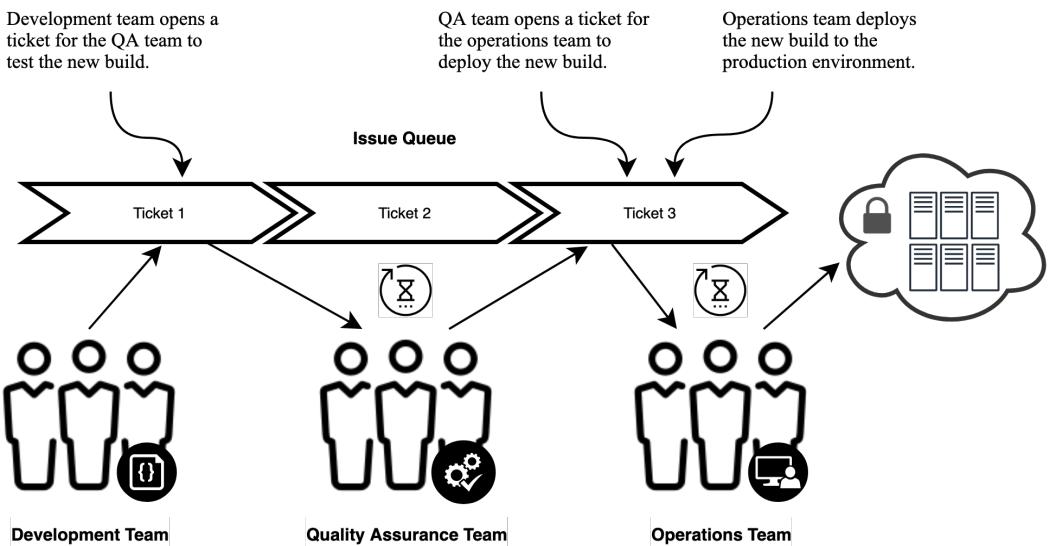


Figure 1.2 In the traditional deployment flow, the development team opens a ticket for the quality assurance team to test a new product version. Once the testing is successful, the QA team opens a ticket for the operations team to deploy the latest version to production.

Thankfully, most development teams compile, test, and produce their deployable artifacts using automated build systems and a process called *Continuous Integration (CI)*. However, the new code's deployment is often a manual process performed by the operations team, involving lengthy manual procedures or partial automation through deployment scripts. In a “worst-case scenario,” the operations engineer would manually copy the executable binary file to the needed location on multiple servers and manually restart the application to make the new binary version take effect. This process is very error-prone and offers few options for controls such as review, approval, auditability, or rollback.

CONTINUOUS INTEGRATION (CI): Continuous Integration involves automated building, testing, and packaging of software applications. In a typical development workflow, software engineers make code changes that are “checked in” to the central code repository. These changes must be tested and “integrated”

with the main code branch intended to be deployed to production. A CI system facilitates the review, building, and testing of code to ensure its quality before merging to the main branch.

With the rise of cloud computing infrastructure, the interfaces to manage compute and network resources have become increasingly based on APIs, allowing for more automation but requiring more programming skills to implement. This, coupled with organizations searching for ways to optimize their operations, reduce deployment times, increase deployment frequency, and improve their computing systems' reliability, stability, and performance, led to a new industry trend: DevOps.

1.1.2 DevOps

DevOps is both an organizational structure and mindset change with an emphasis on automation. Instead of having a separate operations team responsible for its deployment and operation, the application's development team takes on this responsibility.

DEVOPS: DevOps is a set of software development practices that combine software development (Dev) and information technology operations (Ops) to shorten the systems development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives.²

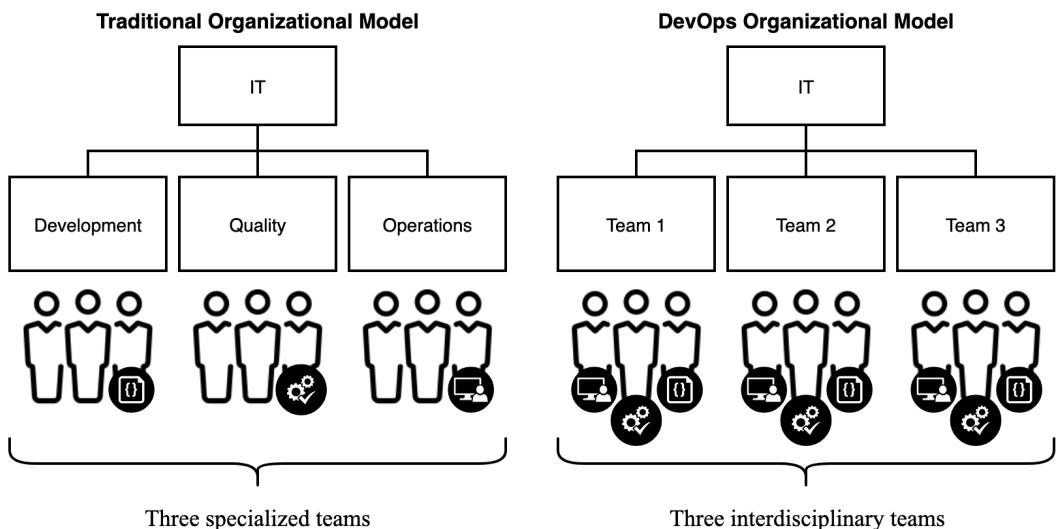


Figure 1.3 The traditional organizational model has separate teams for development, quality, and operations. A DevOps organizational model allows interdisciplinary teams centered around a specific product or component. Each DevOps team is self-sufficient and contains members with the skills to develop, test, and deploy their application.

²<https://en.wikipedia.org/wiki/DevOps>

The above diagram shows how, in a traditional operations model, the organization is split between functional boundaries, with different teams for Development, Quality, and Operations. In the DevOps model, teams are divided between products or components and are interdisciplinary, containing team members with skill sets across all functions. While the diagram above indicates team members with a specific role, members of a high functioning team practicing DevOps will all contribute across functions, each being able to code, test, deploy, and operate their product or component.

The benefits of DevOps include:

- Better collaboration between development and operations
- Improved product quality
- More frequent releases
- Reduced time-to-market for new features
- Decreasing costs of design, development, and operations

Case Study: Netflix.

Netflix was one of the early adopters of the DevOps process, with every engineer responsible for code, test, deploy, and support of his/her features. Netflix's culture also believes in "Freedom and Responsibility," which means every engineer can push releases independently but must ensure the proper operation of that release. All deployment processes are fully automated so that engineers can deploy and rollback with the press of a button. All new features are literally in end-users' hands as soon as the functionality is complete.

1.1.3 GitOps

The term "GitOps" was coined/popularized in August 2017 by a series of blogs by Alexis Richardson, the co-founder and CEO of Weaveworks.³ Since then, it has developed significant mindshare among the cloud-native community in general and Kubernetes in particular. GitOps is a DevOps process characterized by:

- Best practices across deployment, management, and monitoring of containerized applications
- Developer-centric experience for managing applications, where fully automated pipelines/workflows using Git are used for development and operations
- Use of the Git revision control system to track and approve changes to the infrastructure and runtime environment of applications

³ <https://www.weave.works/blog/gitops-operations-by-pull-request>

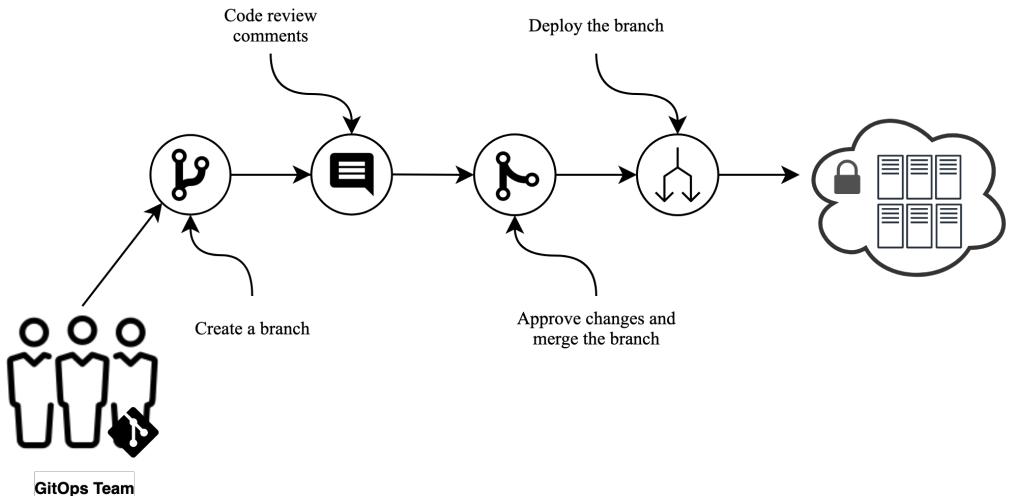


Figure 1.4 The GitOps release workflow starts with creating a branch of the repository containing changes to the definition of the system's desired state. These changes are code reviewed by other team members and, once approved, merged to the main branch. The main branch is processed by a GitOps “operator” that deploys the desired configuration.

GitHub (and Gitlab, BitBucket, etc.) is central to the modern software development lifecycle, so it seems natural that it also be used for system operation and management as well.

When following a GitOps model, the system's desired configuration is stored in a revision control system, such as Git. Instead of making changes directly to the system using a UI or CLI, changes are made to those configuration files representing the desired state. A difference (diff) between the desired state stored in Git and the system's actual state indicates that not all changes have been deployed yet. These changes can be reviewed and approved using standard revision control processes, such as “Pull Request,” “Code Review,” and “Merge to Master.” Once changes have been approved and merged to the main branch, an “operator” software process is responsible for changing the system's current state into the desired state based on the configuration stored in Git.

In an ideal implementation of GitOps, manual changes to the system are not permitted, and all changes to the configuration must be made to files stored in Git. In an extreme case, permission to change the system is only granted to the “operator” software process. The infrastructure and operations engineers' role in a GitOps model shifts from performing the infrastructure changes and application deployments to developing and maintaining the GitOps automation and helping teams review and approve changes using Git.

Git has many features and technical capabilities that make it an ideal choice for use with GitOps.

1. Git stores each commit. With proper access control and security configuration (covered in chapter 6), all changes are auditable and tamper-proof.

2. Each commit in Git represents a complete configuration of the system up to that point in time.
3. Each commit object in Git is associated with its parent commit so that, as branches are created and merged, the commit history is available when needed.

NOTE GitOps is important because it enables traceability of changes made to an environment and enables easy rollback, recoverability, and self-healing using Git, a tool most developers are already familiar with.

Git provides the basis to validate and audit deployments. While it may be possible to implement GitOps using a version control system other than Git, Git's distributed nature, branching and merging strategy, and widespread adoption makes it an ideal choice.

GitOps doesn't require a particular set of tools, only that the tools offer this standard functionality:

- Operate on the desired state of the system that is stored in Git
- Detect differences between the desired state and the actual state
- Perform the required operations on the infrastructure to synchronize the actual state with the desired state

While this book focuses on GitOps in relationship to Kubernetes, many of the principles of GitOps could be implemented independently of Kubernetes.

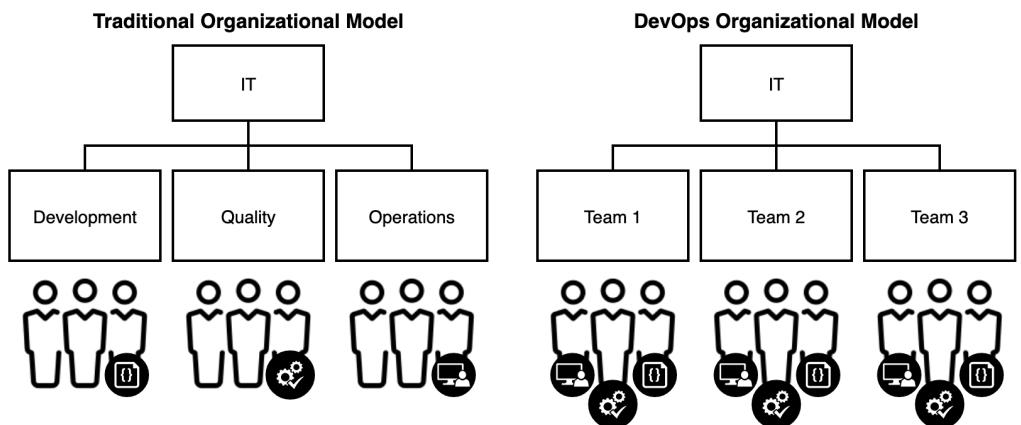


Figure 1.5 In the traditional organizational model, there is a clear separation among development, quality, and operations teams. In the DevOps Organizational Model, each team is responsible for the application's development, quality, and operations.

1.2 Developer Benefits of GitOps

GitOps provides many benefits to developers since it allows them to treat the configuration of infrastructure and deployment of code in a very similar manner to how they manage their software development process and with a familiar tool: Git.

1.2.1 Infrastructure as Code

Infrastructure as Code (IaC) is a foundational paradigm for GitOps. The configuration of the infrastructure that runs your applications is accomplished by executing an automated process rather than manual steps.⁴ In practice, IaC means that infrastructure changes are codified, and the “source code” for the infrastructure are stored in a version control system. Let’s go through the most notable benefits.

Repeatability: Everyone who has experience provisioning infrastructure manually agrees that this is a very time consuming and error-prone process. Don’t forget that the same process has to be repeated multiple times since applications are typically deployed into multiple environments. If a problem is discovered, it is easier to roll back to an earlier working configuration with a repeatable process, allowing quicker recoveries.

Reliability: The automated process significantly reduces the chance of inevitable human errors, thereby reducing the possibility of outages. Once the process is codified, the infrastructure quality no longer depends on the particular engineer’s knowledge and skill who is performing the deployment. The automation of the infrastructure configuration can be steadily improved.

Efficiency: IaC increases the productivity of the team. With IaC, engineers are more productive because they use familiar tools, such as application programming interfaces (APIs), software development kits (SDKs), version control systems, and text editors. Engineers can use familiar processes and can take advantage of code review and automated testing.

Cost-saving: The initial implementation of IaC requires significant effort and time investment. However, despite the initial cost, it is more cost-effective in the long-run. The provisioning of infrastructure for the next environment does not require wasting valuable engineer time for manual configuration. Since provisioning is quick and cheap, there is no need to keep unused environments running. Instead, each environment might be created on-demand and destroyed once no longer needed.

Visibility: When you define Infrastructure as Code, the code itself documents the intention of how the infrastructure should look.

Infrastructure as Code enables developers to produce higher quality software and saves time and money. It might be easier to configure the infrastructure manually for one environment. However, it will become increasingly challenging to maintain that environment, along with dozens of other environments for your application. Using automated infrastructure provisioning and following Infrastructure as Code principles enable repeatable deployments and prevent runtime issues caused by configuration drift or missing dependencies.

1.2.2 Self-Service

As mentioned previously, in a traditional operations model, infrastructure management is performed by a dedicated team or even a separate organization within the company.

However, there is a problem: this approach does not scale. The dedicated team will quickly become a bottleneck, no matter how many members it has. Instead of making an infrastructure change themselves, application developers have to file a ticket, send an email,

⁴<https://www.hashicorp.com/resources/what-is-infrastructure-as-code>

schedule a meeting, and wait. Regardless of the process, there is a barrier that introduces many delays and discourages the team from proactively proposing infrastructure changes.

GITOPS FLOW

GitOps aims to break the barrier by automating the process and making it self-service.

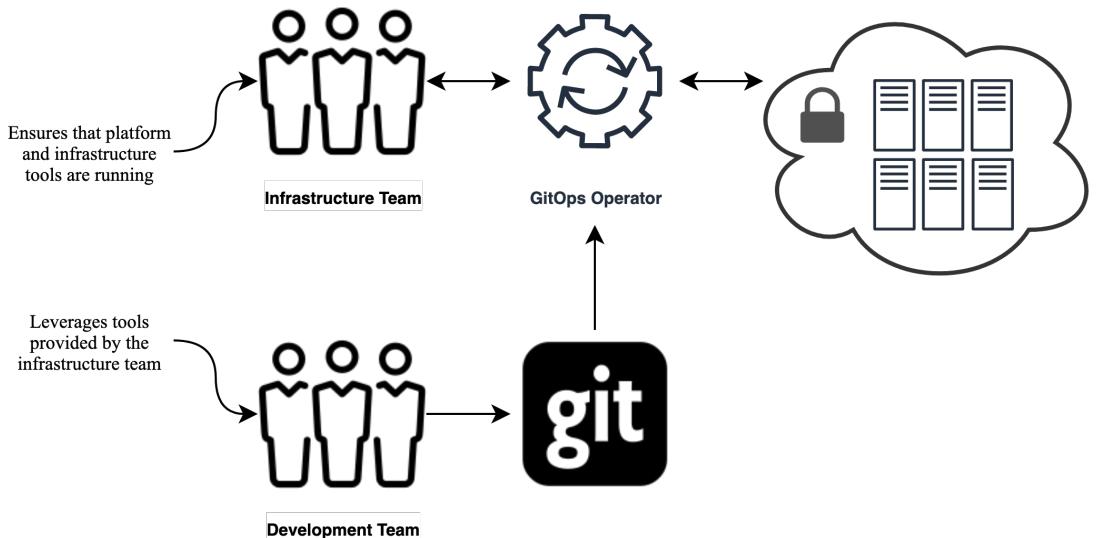


Figure 1.6 The development team can change the system's desired state by updating files stored in the Git repository. These changes are code reviewed by other team members and, once approved, merged to the main branch. The main branch is processed by a GitOps "operator" that deploys the cluster's desired configuration.

Instead of sending a ticket, when using a GitOps model, the developer independently works on a solution and commits a change to the infrastructure's declarative configuration in the repository. The infrastructure change does not require cross-team communication anymore, allowing the application development team to move forward much more quickly and have more freedom to experiment. The ability to make infrastructure changes rapidly and independently encourages developers to take ownership of their application infrastructure. Instead of just asking a central operations team for a solution, developers can experiment and develop a design that efficiently solves the business requirements.

This does not mean that developers get full control to do whatever they want, possibly compromising security or reliability. Every change requires creating a pull request that can be reviewed by another member of the application development team, as described in the next sections.

The advantage of GitOps is that it allows self-service of infrastructure changes and provides the right balance between control and development speed.

1.2.3 Code Reviews

Code review is a software development practice in which code changes are proactively examined for errors or omissions by a “second pair of eyes,” leading to fewer preventable outages. Performing code reviews is a natural process in the software development lifecycle with which software engineers doing DevOps/GitOps should be familiar. Once the DevOps engineer can treat Infrastructure as Code, the logical next step is to perform code reviews on the infrastructure changes before deployment. When using GitOps with Kubernetes, the “code” being reviewed may primarily be Kubernetes YAML manifests or other declarative configuration files, not traditional code written in a programming language.

Besides error prevention, code reviews provide the following additional benefits:

Teaching and sharing knowledge. While reviewing the changes, the reviewer has a chance not just to give feedback but also to learn something new.

Consistency in design and implementation. During the review, the team can ensure changes are aligned with the overall code structure and follow the company code style guidelines.

Team cohesion. Code review is meant not just to criticize and request changes. This is also an excellent way for team members to give kudos to each other, get closer, and make sure everyone is fully engaged.

With a proper code review process, only verified and approved infrastructure changes are committed to the main branch, preventing errors and incorrect modifications to the operating environment. Code review doesn’t necessarily need to be done 100% by humans. The code review process also can run automated tools like code linters⁵, static code analysis, security tools, etc.

NOTE Other automation tools for code and vulnerability analysis will be covered in chapter 4.

Code reviews have long been accepted as a critical part of software development best practices. The key premise of GitOps is that the same rigor of code reviews used on the application code should also be applied to changes to the application operational environment.

1.2.4 Git Pull Requests

The Git version control system provides a mechanism where proposed changes can be committed to a branch or fork and then merged to the main branch through a *pull request*. In 2005 Git introduced a `request-pull` command. The command generates a human-readable summary of all the changes, which can be mailed to the project maintainer manually. The pull request collects all the changes to the repository files and presents the differences for code review and approval.

Pull requests can be used to enforce pre-merge code reviews. Controls can be put in place to require specific testing or approvals before a pull request being merged to the main

⁵ [https://en.wikipedia.org/wiki/Lint_\(software\)](https://en.wikipedia.org/wiki/Lint_(software))

branch. As with code reviews, pull requests are a familiar process in the software development life cycle that software engineers likely already use.

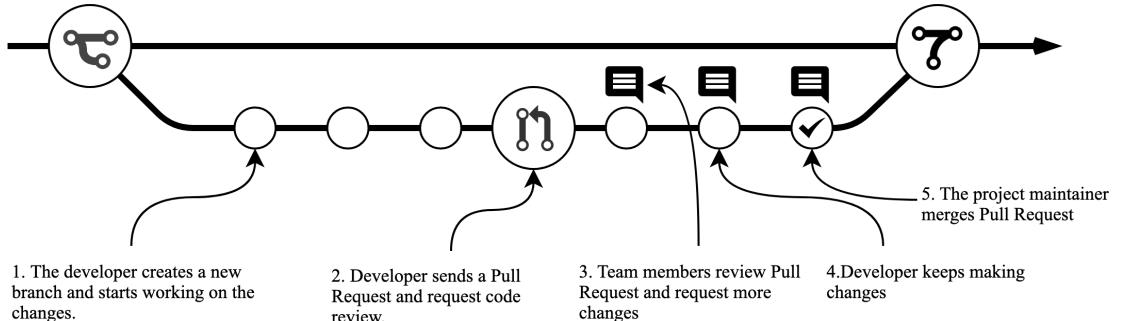


Figure 1.7 The Pull Request life cycle allows multiple rounds of code review and revisions until approval of the changes. Once approved, the change may be merged to the main branch and the PR branch deleted.

Figure 1.7 demonstrates the typical Pull Request life cycle:

1. The developer creates a new branch and starts working on the changes.
2. Once changes are ready, the developer sends a Pull Request and requests code review.
3. Team members review the Pull Request and request more changes (if needed).
4. The developer keeps making changes in a branch until the Pull Request is approved.
5. The project maintainer merges Pull Request into the main branch.
6. Once merged, the branch used for the PR may be deleted.

The review step is especially interesting when applied to an infrastructure change review. After the Pull Request is created, the project maintainers receive a notification and review the proposed changes. As a result, reviewers ask questions, receive answers, and possibly request more changes. That information is typically stored and available for future reference. That means that now the Pull Request is a live documentation of an infrastructure change. In case of an incident, it is straightforward to find who made the change and why it was applied.

1.3 Operational Benefits of GitOps

Combining a GitOps methodology with Kubernetes' declarative configuration and active reconciliation model provides many operational benefits that aim to provide a more predictable and reliable system.

1.3.1 Declarative

One of the more prominent paradigms to emerge from the DevOps movement is the model of *declarative systems and configuration*. Simply put, with declarative models, you describe *what* you want to be achieved, as opposed to *how* to get there. This is in contrast to an

imperative model, in which you describe a sequence of instructions to manipulate the state of the system to reach your desired state.

To illustrate this difference, imagine the following two styles of a television remote control: an *imperative* style and a *declarative* style. Both remotes can control the TV's power, volume, and channel. For the sake of discussion, assume that the TV has only three volume settings (loud, soft, mute) and only three channels (1, 2, 3).

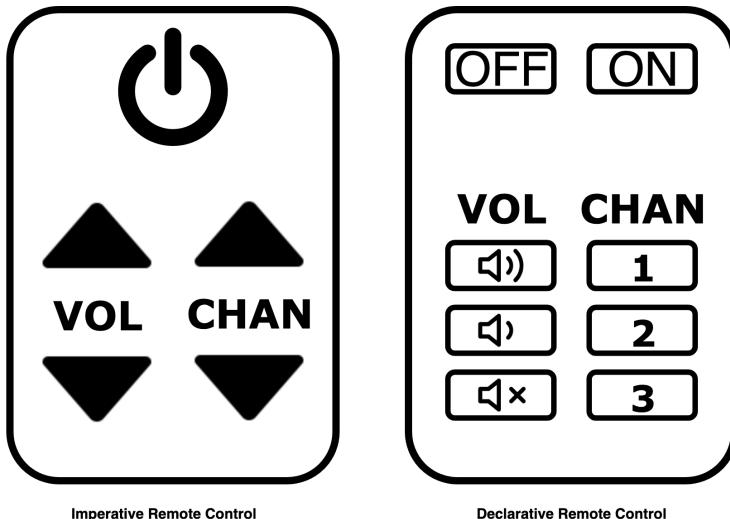


Figure 1.8 This figure illustrates the differences between an imperative vs. declarative remote control. The imperative remote lets you perform operations like “increment the channel by one” or “toggle the state of the power.” In contrast, the declarative remote lets you perform operations like “tune to channel 2” or “set the state of the power to off.”

IMPERATIVE REMOTE EXAMPLE

Suppose you had the simple task of changing to channel 3 using both remotes. To accomplish this task using the imperative remote, you would use the “channel up” button, which signals the TV to increment the current channel by one. To reach channel 3, you would keep pressing the “channel up” button some number of times until the TV reaches the desired channel.

DECLARATIVE REMOTE EXAMPLE

Contrast this to the declarative remote, which provides individual buttons that jump directly to the specific, numbered channel. In this case, to switch to channel 3, you would press the “channel 3” button once, and the TV would be on the correct channel. You are declaring your intended end state (I want the TV tuned to channel 3). Whereas with the imperative remote, you describe the actions needed to be performed to achieve your desired state (keep pressing “channel up” until the TV is tuned to channel 3).

You may have noticed that in the imperative approach of changing channels, the user must consider whether or not to continue pushing the “channel up” button, depending on to which channel the TV is currently tuned. Whereas in the declarative approach, you can press the “channel 3” button without a second thought. This is because the “channel 3” button in the declarative remote is considered to be “idempotent,” whereas the “channel up” button in the imperative remote is not.

IDEMPOTENCY: Idempotency is a property of an operation, whereby the operation can be performed any number of times and still produce the same result. In other words, an operation is said to be idempotent if you can perform the operation an arbitrary number of times, and the system is in the same state as it would be if you had performed the operation only once. Idempotency is one of the properties distinguishing declarative systems from imperative systems. Declarative systems are idempotent; imperative systems are not.

1.3.2 Observability

Observability is the ability to examine and describe a system's current running state and be alerted when unexpected conditions occur. Deployed environments are expected to be observable. In other words, you should always be able to inspect an environment to see what is currently running and how things are configured. To that end, service and cloud providers will provide a fair number of methods to promote observability (CLIs, APIs, GUIs, dashboards, alerts & notifications), making it as convenient as possible for users to understand the current state of their environment.

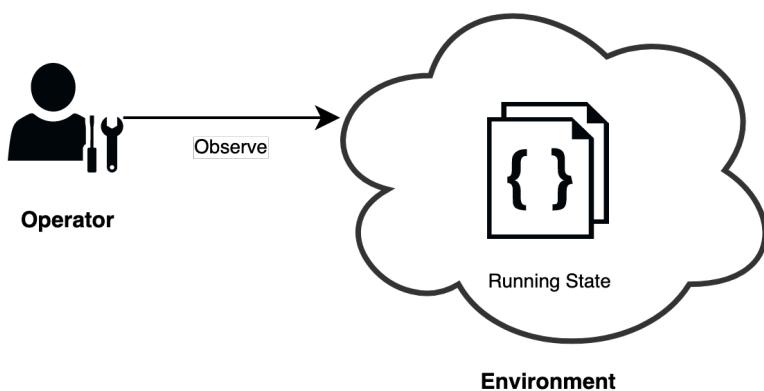


Figure 1.9 Observability is the operator's ability (perhaps human or automated) to determine an environment's running state. The operator can make informed decisions on what changes to the environment are needed only once the environment's current running state is known. Proper control of an environment requires observability of that environment.

Although these observability mechanisms can help answer the question of “what's currently running in my environment?” they cannot answer the question of “for the resources currently

configured and running in my environment, are they *supposed* to be configured and running in that way?" If you have ever held duties as a system administrator or operator, you're likely all too familiar with this problem. At one point or another, typically when troubleshooting an environment, you come across a suspect configuration setting and wonder to yourself: this doesn't seem right — did someone (possibly yourself) accidentally or mistakenly change this setting? Or is the setting intentional?

In all likelihood, you may already be practicing a fundamental principle of GitOps, which is to store a copy of your application configuration in source control and use it as a "source of truth" of the desired state of your application. You might not be storing it in Git to drive continuous deployment, but simply to have a copy of the configuration duplicated somewhere so that the environment can be reproduced (e.g., in a disaster recovery scenario). This copy can be thought of as the desired application state and, aside from the disaster recovery use case, serves another useful purpose. It enables operators to compare the actual running state to the desired state held in source control at any point in time and verify that they match.

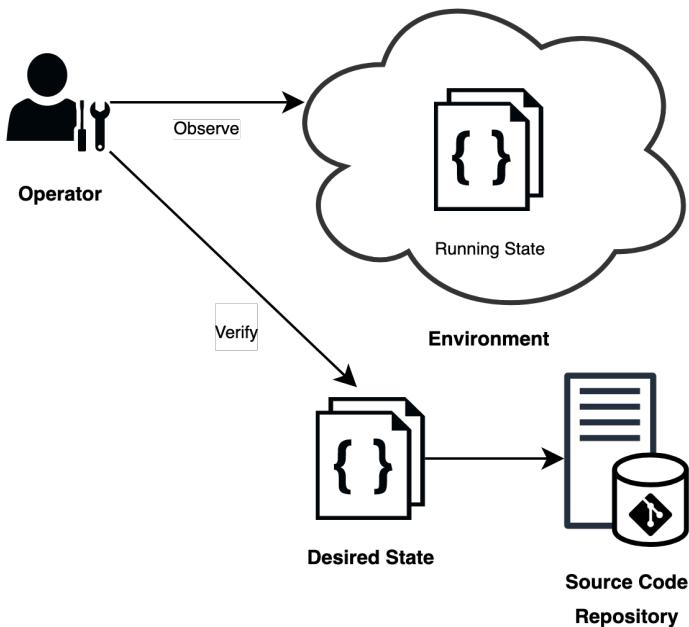


Figure 1.10 If the environment's running state can be observed, and the desired state of the environment is defined in Git, the environment can be verified by comparing the two.

The ability to verify your environment is a core tenet to GitOps, which has been formalized as a practice. By storing your desired state into one system (i.e., Git) and regularly comparing that desired state with the running state, you unlock an entirely new dimension of

observability. Not only do you have the standard observability mechanisms provided by your provider, but you are now able to detect divergence from the desired state.

Divergence from your desired state, also called *configuration drift*, can happen for any number of reasons. Common examples include mistakes made by operators, unintended side-effects due to automation, error scenarios. It could even be expected, like a temporary state caused by some transition period (e.g., maintenance mode).

But the most significant reason why there might be a divergence in configuration could be malicious. In the worst case, a bad actor could have compromised the environment and reconfigured the system to run a malicious image. For this reason, observability and verification are crucial for the security of the system. Without a source of truth of your desired state put in place, and without a mechanism to verify convergence to that source of truth, it is impossible to know that your environment is truly secure.

1.3.3 Auditability & Compliance

Allowing for *compliance and auditability* is a must for organizations doing business in countries with laws and regulations affecting information management and frameworks for assessing compliance, which is most countries in this day and age. Some industries are more regulated than others, but almost all companies need to comply with basic privacy and data security laws. Many organizations have to invest substantially in their processes and systems to be compliant and auditable. With GitOps and Kubernetes, most of the compliance and auditability requirements can be satisfied with minimal effort.

Laws and regulations affecting information management and frameworks for assessing compliance

Compliance: Compliance refers to verifying that an organization's information system meets a particular set of industry standards, typically focused on customer data security and the adherence to the organization's documented policies about the people and systems that have access to that customer data. Chapter 6 will cover access control in-depth, and chapter 4 will cover pipelines to define and enforce your deployment process for compliance.

Auditability: Auditability is the ability of a system to be verified as being compliant with a set of standards. If a system can't be shown to an internal or external auditor to be compliant, then no statement about the system's compliance can be made. Chapter 8 will cover "Observability," including using Git commit history and Kubernetes events for auditability.

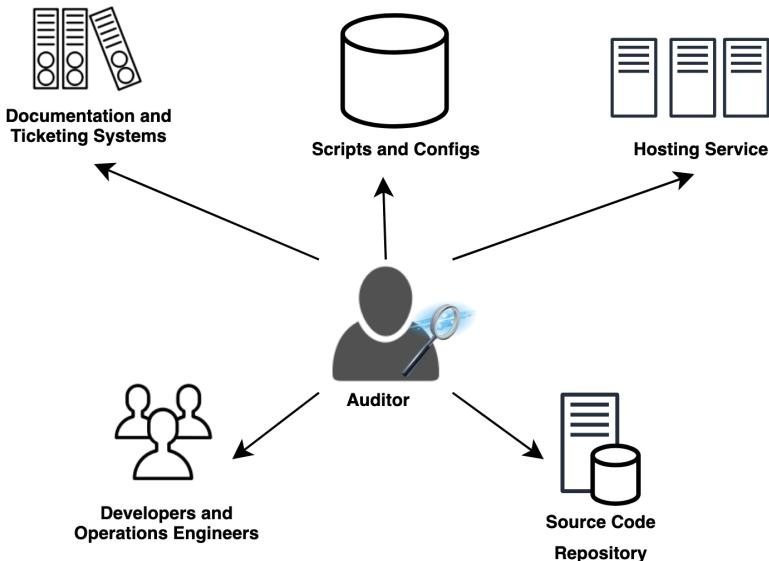


Figure 1.11 With a traditional audit process, it is often difficult to determine the system's desired state. Auditors may need to look at various sources for this information, including documentation, change requests, and deployment scripts.

Compliance describes the ability to act according to an order, a set of rules, or requests. Corporate compliance is the process of making sure your company and employees follow external regulations that apply to your organization, such as federal and state laws, regulations, standards, and ethical practices. Chapter 4 will discuss in detail how this is accomplished with GitOps and Kubernetes.

Effective corporate compliance will encompass both internal systems of control and policies imposed to achieve compliance with external rules. Enforcing compliance in corporate policy will help your company prevent and detect violations of regulations. The compliance process needs to be continuous to avoid fines and lawsuits for your organization.

Case Study: Facebook and Cambridge Analytica.

Cambridge Analytica, a political data firm hired by President Trump's 2016 election campaign, gained improper access to the private information of more than 50 million Facebook users. The data was used to generate a personality score for each user and match it with U.S. voter records. Cambridge Analytica then used this information for its voter profiling and targeted advertising services. Facebook was found not to have implemented the proper controls required to enforce data privacy and was eventually fined \$5 billion by FTC due to the breach.⁶

Auditability refers to an auditor's ability to achieve a comprehensive examination of an organization's internal controls. In a typical audit, the auditor will request evidence to ensure rules and policies are enforced accordingly.

⁶<https://www.ftc.gov/news-events/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions>

Evidence could include the process of restricting access to user data, the handling of Personal Identifiable Information (PII), and the integrity of the software release process.

Case Study: Payments Card Industry Data Security Standards

The Payments Card Industry Data Security Standards (PCI DSS) is an information security standard for organizations that handle branded credit cards from the major card schemes. Violation of the PCI DSS could result in steep fines and, worse case, suspension from credit card processing. For example, PCI DSS dictates that “Access control systems are configured to enforce privileges assigned to individuals based on job classification and function.” During an audit, organizations need to provide evidence that access control systems are in place for PCI compliance.⁷

For *Software as a Service* (SaaS) companies (e.g., Salesforce, Google) and cloud-computing providers (e.g., Google Cloud Platform, AWS), information security is a significant concern since these third-party vendors handle critical business operations for other enterprises. Any mishandled data can leave enterprises vulnerable to attacks, such as data theft, extortion, and malware installation. Service Organization Control 2 (SOC 2) is an auditing procedure that ensures your service providers securely manage your data to protect your organization's interests and the privacy of its clients. For security-conscious businesses, SOC 2 compliance is a minimal requirement when considering a SaaS provider. SOC 2 consists of five Trust Service Principles.

- **Security** - The system is protected against unauthorized access, both physical and logical
- **Availability** - The system is available for operation and use as committed or agreed
- **Processing Integrity** - System processing is complete, accurate, timely, and authorized
- **Confidentiality** - Information designated as confidential is protected as required
- **Privacy** - Personal information is collected, used, retained, disclosed, and destroyed in conformity with the commitments in the entity's privacy notice and with the criteria outlined in Generally Accepted Privacy Principles (GAPP)

So what does all that have to do with GitOps?

Git is a version control software that helps organizations manage changes and access control to their code. Git keeps track of every modification to the code in a special kind of database designed to preserve the managed source code's integrity. The files' content and the true relationships between files and directories, versions, tags, and commits in the Git repository are secured with the SHA checksum hashing algorithm. This protects the code and the change history against both accidental and malicious change and ensures that the history is fully traceable.

Because of the hashing algorithm, Git can guarantee the authentic content history of your source code. Other version control systems may not have protection against the secret alteration of source code. Without the authenticity guarantee, the organization will have difficulty monitoring and controlling their code changes' integrity.

⁷https://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard

Git also maintains the complete long-term change history of every file in the repository. This means every change made by individual users over the life of the repository, including file creation, file deletion, and edits to file contents, is tracked and attributed. Different version control tools differ on how well they handle the renaming and moving of files. Git's history tracking also includes the author, date, and written notes on each change's purpose. With well-written commit comments, you also know why a particular commit was made. Git can also integrate with project management and bug tracking software, allowing full traceability of all changes and enabling root cause analysis and other forensics.

As mentioned earlier, Git supports the Pull Request mechanism, which can enforce no single person can alter the system without approval by a second person. Once the Pull Request is approved, changes are recorded in the secured Git change history. Git's strength in change control, traceability, and change history authenticity, along with Kubernetes' declarative configuration, naturally satisfy the Security, Availability, and Processing integrity principles of SOC 2. Evidence can be easily generated from Git and Kubernetes for any SOC 2 audit process.

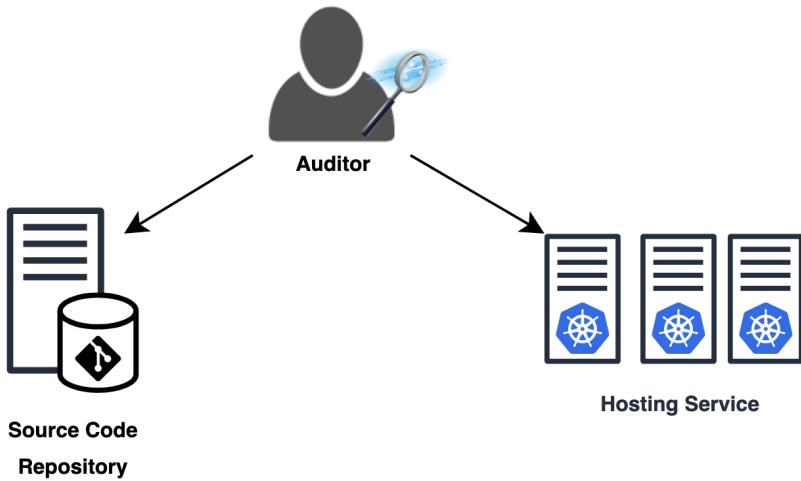


Figure 1.12 With GitOps, the audit process can be simplified since the auditor can determine the system's desired state by examining the source code repository. The current state of the system can be determined by reviewing the hosting service and Kubernetes objects.

1.3.4 Disaster Recovery

Disasters happen for many reasons and take many forms. They might be naturally occurring (an earthquake hitting a datacenter), caused by an equipment failure (loss of hard drives in a storage array), accidental (a software bug corrupting a critical database table), or even malicious (cyber-attack causing data loss).

GitOps aids in the recovery of infrastructure environments by storing these declarative specifications of the environment under source control as a source of truth. Having a

complete definition of what the environment should facilitate the re-creation of the environment in the event of a disaster. Disaster recovery becomes a simple exercise of (re)applying all the configuration stored in the Git repository. Using this method, one might observe that there is not much of a difference between the procedures during a disaster versus routine day-to-day upgrades and deployments. With GitOps, you are, in effect, practicing disaster recovery procedures on a regular basis, making you well prepared on the occasion that a real disaster strikes.

IMPORTANCE OF DATA BACKUP: While GitOps helps simplify the DR story for computing and networking infrastructure, recovery of persistent and stateful applications do need to be handled differently. There is no substitute for traditional DR solutions for storage-related infrastructure, namely backups, snapshotting, and replication.

1.4 Summary

- GitOps is a DevOps deployment process using Git as the system of record to manage deployment for complex systems.
- Traditional Ops requires a separate team for deployment, and a new version can take days (if not weeks) to get deployed.
- DevOps enables engineers to deploy a new version as soon as the code is complete without waiting for a centralized operations team.
- GitOps provides full traceability and release control.
- Declarative models describe what you want to be achieved instead of the steps necessary to achieve it.
- Idempotency is a property of an operation, whereby the operation can be performed any number of times and still produce the same result.
- Additional GitOps benefits include:
 - a) Pull Request for code quality and release control
 - b) Observable running state and desired state
 - c) Simplify compliance and auditability process with historical authenticity and traceability
 - d) More straightforward disaster recovery and rollback procedures consistent with the familiar deployment experience

2

Kubernetes & GitOps

This chapter covers:

- Problem-solving with Kubernetes
- Running and managing Kubernetes locally
- The basics of GitOps
- Implementing a simple Kubernetes GitOps operator

In chapter 1, you learned about Kubernetes and why its declarative model makes it an excellent match to be managed using GitOps. This chapter will briefly introduce Kubernetes architecture and objects and learn about the differences between declarative and imperative object management. By the end of this chapter, you will implement a basic GitOps Kubernetes deployment operator.

2.1 Kubernetes Introduction

Before diving into why Kubernetes and GitOps work so well together, let's talk about Kubernetes itself. This section provides a high-level overview of Kubernetes, how it compares to other container orchestration systems, and its architecture. We will also have an exercise that demonstrates how to run Kubernetes locally, which will be used for the other exercises in this book. This section is only a brief introduction and refresher to Kubernetes. For a fun but informative overview of Kubernetes, check out "The Illustrated Children's Guide to Kubernetes" and "Phippy Goes to the Zoo" by the Cloud Native Computing Foundation⁸. If you are completely new to Kubernetes, we recommend reading *Kubernetes in Action*, by Marko Lukša and then returning to this book. If you are already familiar with Kubernetes and running minikube, you may skip to the exercise at the end of section 2.1.

⁸<https://www.cncf.io/phippy/>

2.1.1 What Is Kubernetes?

Kubernetes is an open-source container orchestration system released in 2014. OK, but what are containers, and why do you need to orchestrate them?

Containers provide a standard way to package your application's code, configuration, and dependencies into a single resource. This enables developers to ensure that the application will run properly on any other machine regardless of any customized settings that machine may have that could differ from the machine used for writing and testing the code. Docker simplified and popularized containerization, which is now recognized as a fundamental technology used to build distributed systems.

NOTE *chroot* An operation available in Unix operating systems, which changes the apparent root directory for the current running process and its children. chroot provides a way to isolate a process and its children from the rest of the system. It was a precursor to containerization and Docker.⁹

While Docker solved the packaging and isolation problem of individual applications, there were still many questions of how to orchestrate the operation of multiple applications into a working distributed system:

- How do containers communicate with each other?
- How is traffic routed between containers?
- How are containers scaled up to handle additional application load?
- How is the underlying infrastructure of the cluster scaled-up to run the required containers?

All these operations are the responsibility of a container orchestration system and are provided by Kubernetes. Kubernetes helps to automate the deployment, scaling, and management of applications using containers.

NOTE *Borg* Borg is Google's internal container cluster-management system used to power online services like Google search, Gmail, and YouTube. Kubernetes leverages Borg's innovations and lessons-learned, explaining why it is more stable and moving so much more quickly than its competitors.¹⁰

Kubernetes was initially developed and open-sourced by Google based on a decade of experience with container orchestration using Borg, Google's proprietary cluster management system. Because of this, Kubernetes is relatively stable and mature for a system so complex. Because of its open API and extendable architecture, Kubernetes has developed an extensive community around it, which has further fueled its success. It is one of the top GitHub projects (as measured by "stars"), provides excellent documentation, and has a significant Slack and Stack Overflow community. There is an endless number of blogs and presentations from community members who share their knowledge of using Kubernetes. Despite being started by Google, Kubernetes is not influenced by a single vendor. This makes the community open, collaborative, and enables innovation.

⁹ <https://en.wikipedia.org/wiki/Chroot>

¹⁰ <https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/>

2.1.2 Other Container Orchestrators

Since late-2016, Kubernetes has become recognized as the dominant de-facto industry-standard container orchestration system in much the same way as Docker has become the standard for containers. However, several Kubernetes alternatives address the same container orchestration problem as Kubernetes. Docker Swarm is Docker's native Container Orchestration Engine that was released in 2015. It is tightly integrated with the Docker API and uses a YAML-based deployment model called Docker Compose. Apache Mesos was officially released in 2016 (although it has a history well before then) and supports large clusters, scaling to thousands of nodes.

While it may be possible to apply a GitOps-approach to deploying applications using other container orchestration systems, this book will focus on Kubernetes.

2.1.3 Kubernetes Architecture

By the end of this chapter, you will complete an exercise that implements a basic GitOps continuous deployment operator for Kubernetes. But to understand how a GitOps operator functions, it is essential that you first understand a few Kubernetes core concepts and learn how it is organized at a high level.

Kubernetes is an extensive and robust system with many different types of resources and operations that can be performed on those resources. Kubernetes provides a layer of abstraction over the infrastructure and introduces the following set of basic objects which represent the desired cluster state:

- Pod - a group of containers that are deployed together on the same host. The Pod is the smallest deployable unit on a Node and provides a way to mount storage, set environment variables, and provide other container configuration information. When all the Containers of a Pod exit, the Pod dies also.
- Service - an abstraction that defines a logical set of Pods and a policy to access them.
- Volume - represents a directory accessible to containers running in a Pod.

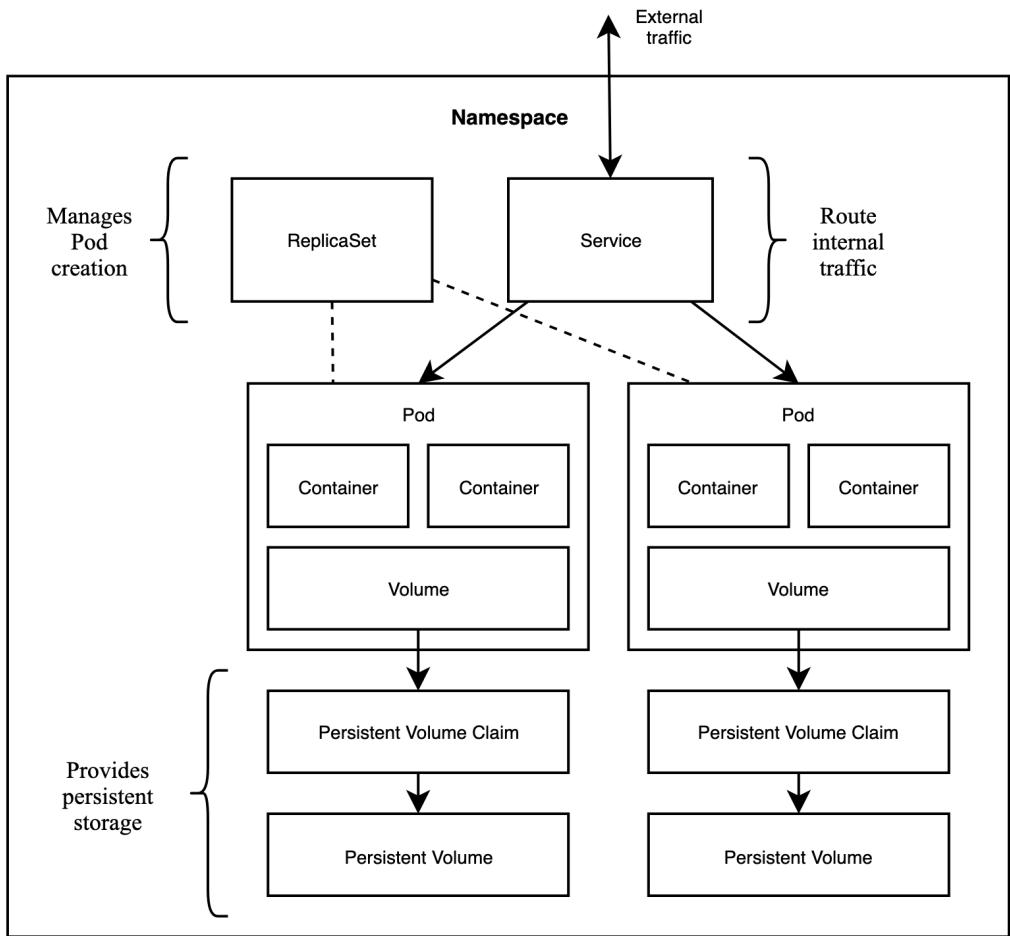


Figure 2.1 This diagram illustrates a typical Kubernetes environment deployed in a Namespace. A ReplicaSet is an example of a higher-level resource that manages the lifecycle of Pods, which is a lower-level, primary resource.

Kubernetes architecture uses primary resources as a foundational layer for a set of higher-level resources. The higher-level resources implement features needed for real production use-cases that leverage/extend the primary resources' functionality. In figure 2.1, you see that the ReplicaSet resource controls the creation of one or more Pod resources. Some other examples of high-level resources include:

- **ReplicaSet** - defines that a desired number of identically configured Pods are running. If a Pod in the ReplicaSet terminates, a new Pod will be started to bring the number of running Pods back to the desired number.

- Deployments - enables declarative updates for Pods and ReplicaSets.
- Jobs - creates one or more pods that run to completion.
- CronJobs - creates Jobs on a time-based schedule

Another important Kubernetes resource are Namespaces. Most kinds of Kubernetes resources belong to one (and only one) Namespace. A namespace defines a naming scope where resources within a particular namespace must be uniquely named. Namespaces also provide a way to isolate users and applications from each other through role-based access controls (RBAC), network policies, and resource quotas. These controls allow creating a multi-tenant Kubernetes cluster where multiple users share the same cluster and avoid impacting each other (e.g., the “noisy neighbor” problem). As we will see in chapter 3, Namespaces are also essential in GitOps for defining application environments.

Kubernetes objects are stored in a control-plane¹¹, which monitors the cluster state, makes changes, schedules work, and responds to events. To perform these duties, each Kubernetes control-plane runs the following three processes:

- kube-apiserver - An entry-point to the cluster providing a REST API to evaluate and update the desired cluster state.
- kube-controller-manager - Daemon continuously monitoring the shared state of the cluster through the API server to make changes attempting to move the current state towards the desired state.
- kube-scheduler - A component that is responsible for scheduling the workloads across the available nodes in the cluster.
- etcd - A highly-available key-value database typically used as Kubernetes' backing store for all cluster configuration data

The actual cluster workloads run using the compute resources of Kubernetes nodes. A node is a worker machine (either a VM or physical machine) that runs the necessary software to allow it to be managed by the cluster. Similar to the masters, each node runs a predefined set of processes:

- kubelet - the primary "node agent" that manages the actual containers on the node.
- kube-proxy - a network proxy that reflects services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding.

¹¹ <https://kubernetes.io/docs/concepts/#kubernetes-control-plane>

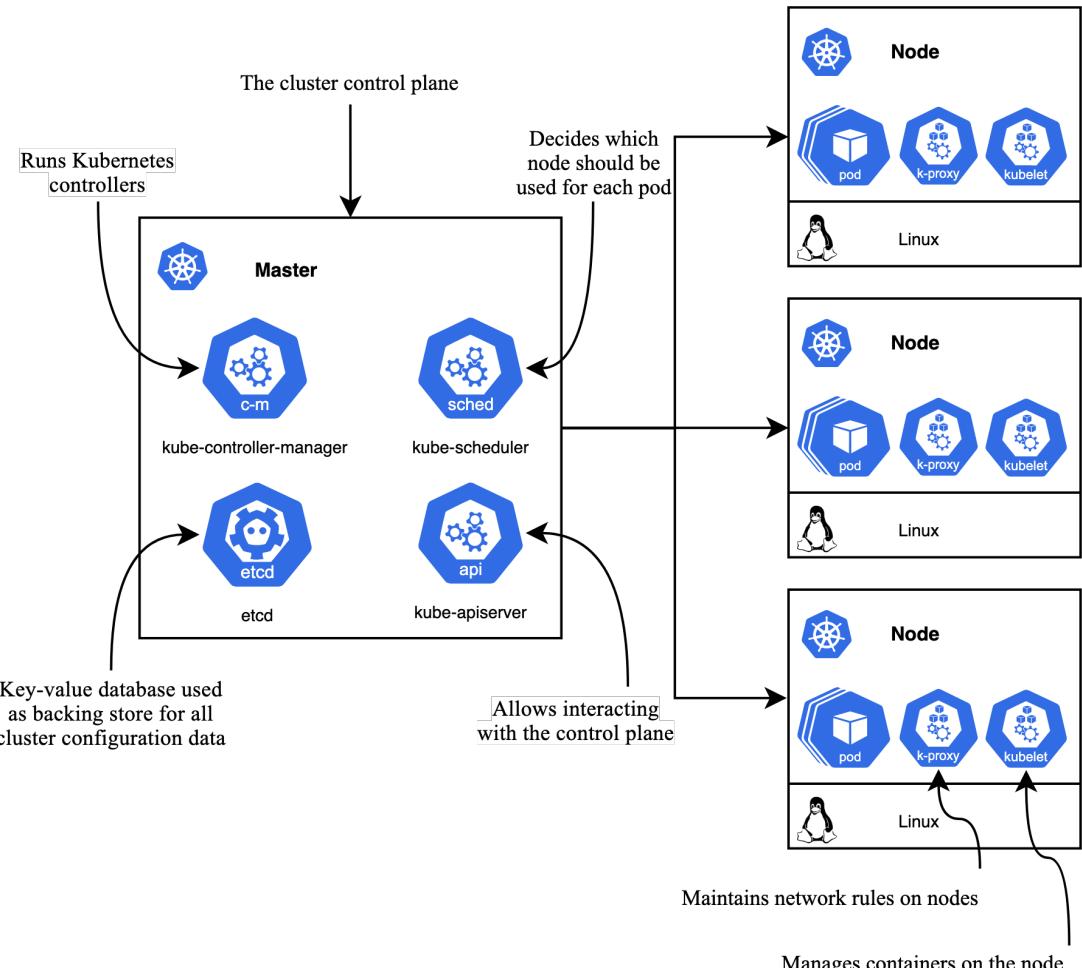


Figure 2.2 A Kubernetes cluster consists of several services that run on the master nodes of the control plane and several other services that run on the cluster's worker nodes. Together, these services provide the essential services that make up a Kubernetes cluster.

2.1.4 Deploying to Kubernetes

In this exercise, you will deploy a website using nginx on Kubernetes. You will review some basic Kubernetes operations and become familiar with minikube, the single-node Kubernetes environment you will use for most exercises in this book.

NOTE *Kubernetes Test Environment – minikube* Refer to Appendix A to set up a Kubernetes test environment using minikube to complete this exercise.

CREATE A POD

As was mentioned earlier in the chapter, a Pod is the smallest object in Kubernetes and represents a particular application workload. A Pod represents a group of related containers running on the same host and having the same operating requirements. All containers of a single pod share the same network address, port space, and (optionally) file system using Kubernetes volumes.

NOTE NGINX NGINX is an open-source software web server used by many organizations and enterprises to host their web sites because of its performance and stability.

In this exercise, you will create a Pod that hosts a website using nginx. In Kubernetes, objects can be defined by a yaml text file “manifest” that provides all the information needed for Kubernetes to create and manage the object. Here is the listing for our nginx Pod manifest.

Listing 2.1 nginx Pod Manifest ([nginx-pod.yaml](#)).

```

kind: Pod                                     #A
apiVersion: v1
metadata:
  name: nginx                                #B
spec:
  restartPolicy: Always
  volumes:
    - name: data                               #D
      emptyDir: {}
  initContainers:
    - name: nginx-init                         #E
      image: docker/whalesay
      command: [sh, -c]
      args: [echo "<pre>$(<pre>$(cowsay -b 'Hello Kubernetes')</pre>) > /data/index.html"]
      volumeMounts:
        - name: data
          mountPath: /data
  containers:
    - name: nginx                               #F
      image: nginx:1.11
      volumeMounts:
        - name: data
          mountPath: /usr/share/nginx/html

```

#A The field `kind` and `apiVersion` are present in every Kubernetes resource and determine what type of object should be created and how it should be handled.

#B In this example, `metadata` has a `name` field that helps identify each Kubernetes resource. Metadata may also contain UID, labels, and other fields that will be covered later.

#C The `spec` section contains configuration which is specific to a particular `kind` of object. In the `pod` example, `spec` includes a list of `containers`, volume shared between containers, and defines the Pod's `restartPolicy`.

#D The volume that is used to share data between containers.

#E The init section contains HTML generated using the `cowsay`¹² command.

#F The main container that serves the generated HTML file using the NGINX server.

¹² <https://en.wikipedia.org/wiki/Cowsay>

You are welcome to type in the above listing and save it with a filename of `nginx-pod.yaml`. However, since this book's object isn't to improve your typing skills, we recommend cloning our public Git repository mentioned in chapter 1 that contains all the listings in this book and using those files directly.

<https://github.com/gitopsbook/resources>

Let's go ahead and start a Minikube cluster and create the NGINX Pod using the following commands:

```
$ minikube start
      (minikube/default)
🕒 minikube v1.1.1 on darwin (amd64)
🔥 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
🔧 Configuring environment for Kubernetes v1.14.3 on Docker 18.09.6
🌐 Pulling images ...
🚀 Launching Kubernetes ...
🛠 Verifying: apiserver proxy etcd scheduler controller dns
🎉 Done! kubectl is now configured to use "minikube"
$ kubectl create -f nginx-pod.yaml
pod/nginx created
```

Here is a diagram of what the Pod looks like running inside minikube:

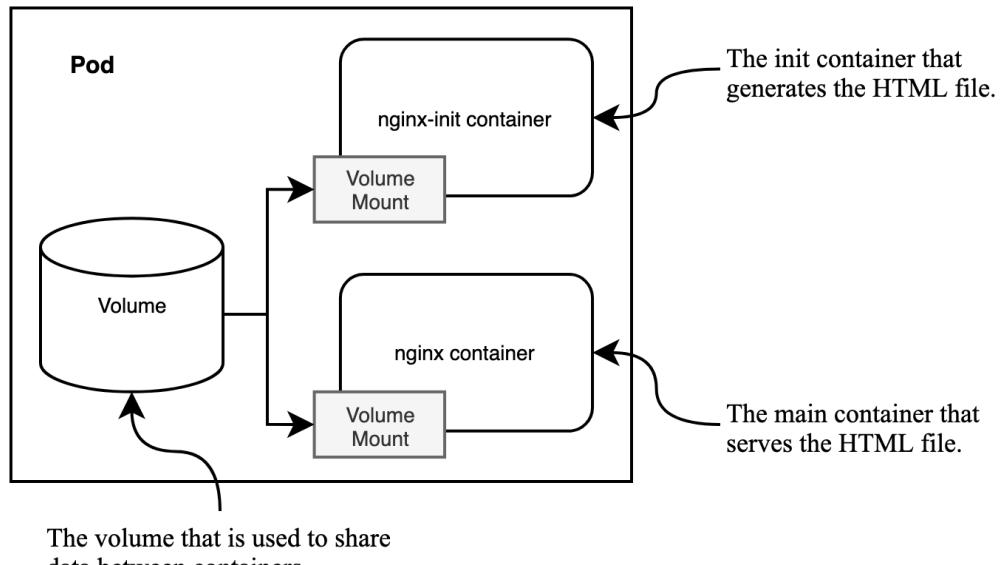


Figure 2.3 The `nginx-init` container writes the desired `index.html` file to the mounted volume. The main `nginx` container also mounts the volume and displays the generated `index.html` when receiving HTTP requests.

GET POD STATUS

As soon as the Pod is created, Kubernetes inspects the spec field and attempts to run the configured set of containers on an appropriate node in the cluster. The information about progress is available in the pod manifest in the `status` field. The `kubectl` utility provides several commands to access it. Let's try to get the pod status using the `kubectl get pods` command.

```
$ kubectl get pods
NAME    READY    STATUS    RESTARTS   AGE
nginx  1/1     Running   0          36s
```

The `get pods` command provides a list of all the pods running in a particular namespace. In this case, we didn't specify a namespace, so it gives the list of Pods running in the default namespace. Assuming all goes well, the `nginx` Pod should be in the "Running" state.

To learn even more about a Pod's status or debug why the Pod is not in the "Running" state, the `kubectl describe pod` command outputs detailed information, including related Kubernetes events.

```
$ kubectl describe pod nginx
Name:          nginx
Namespace:     default
Priority:      0
Node:          minikube/192.168.99.101
Start Time:    Sat, 26 Oct 2019 21:58:43 -0700
Labels:        <none>
Annotations:   kubectl.kubernetes.io/last-applied-configuration:
               {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{},"name":"nginx","namespace":"default"},"spec":{"containers":[{"image":"nginx:1..."}]}
Status:        Running
IP:            172.17.0.4
Init Containers:
  nginx-init:
    Container ID:   docker://128c98e40bd6b840313f05435c7590df0eacf6ce989ec15cb7b484dc60d9bc
    Image:          docker/whalesay
    Image ID:       docker-pullable://docker/whalesay@sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
    Port:           <none>
    Host Port:     <none>
    Command:
      sh
      -c
    Args:
      echo "<pre>$ (cowsay -b 'Hello Kubernetes')</pre>" > /data/index.html
    State:          Terminated
      Reason:       Completed
      Exit Code:    0
      Started:     Sat, 26 Oct 2019 21:58:45 -0700
      Finished:    Sat, 26 Oct 2019 21:58:45 -0700
    Ready:         True
    Restart Count: 0
    Environment:   <none>
    Mounts:
```

```

    /data from data (rw)
    /var/run/secrets/kubernetes.io/serviceaccount from default-token-vbhsd (ro)
Containers:
  nginx:
    Container ID: docker://071dd94670958003b728cef12a5d185660d929ebfeb84816dd060167853e245
    Image:          nginx:1.11
    Image ID:      docker-
                  pullable://nginx@sha256:e6693c20186f837fc393390135d8a598a96a833917917789d63766cab6c5
                  9582
    Port:          <none>
    Host Port:    <none>
    State:        Running
      Started:   Sat, 26 Oct 2019 21:58:46 -0700
    Ready:        True
    Restart Count: 0
    Environment:  <none>
    Mounts:
      /usr/share/nginx/html from data (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-vbhsd (ro)
Conditions:
  Type        Status
  Initialized  True
  Ready       True
  ContainersReady  True
  PodScheduled  True
Volumes:
  data:
    Type:      EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
    SizeLimit: <unset>
  default-token-vbhsd:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-vbhsd
    Optional:   false
QoS Class:  BestEffort
Node-Selectors: <none>
Tolerations:  node.kubernetes.io/not-ready:NoExecute for 300s
               node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type  Reason     Age   From           Message
  ----  -----     ---  ----           -----
  Normal Scheduled  37m   default-scheduler  Successfully assigned default/nginx to
                                minikube
  Normal Pulling    37m   kubelet, minikube  Pulling image "docker/whalesay"
  Normal Pulled     37m   kubelet, minikube  Successfully pulled image "docker/whalesay"
  Normal Created    37m   kubelet, minikube  Created container nginx-init
  Normal Started    37m   kubelet, minikube  Started container nginx-init
  Normal Pulled     37m   kubelet, minikube  Container image "nginx:1.11" already present
                                on machine
  Normal Created    37m   kubelet, minikube  Created container nginx
  Normal Started    37m   kubelet, minikube  Started container nginx

```

Typically the Events section will contain clues as to why a Pod is not in the “Running” state.

The most exhaustive information is available via `kubectl get pod nginx -o=yaml`, which outputs the full internal representation of the object in YAML format. The raw YAML output is difficult to read, and it is typically meant for programmatic access by resource

controllers. Kubernetes resource controllers will be covered in more detail later in this chapter.

ACCESSING THE POD

A Pod in “Running” state means that all containers successfully started, and the nginx Pod is ready to serve requests. If the nginx pod in your cluster is running, then we can try accessing it and prove that it is working.

Pods are not accessible from outside the cluster by default. There are multiple ways to configure external access, which include Kubernetes services, ingress, and more. For the sake of simplicity, we are going to use the command `kubectl port-forward` that forwards connections from a local port to a port on a pod:

```
$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Keep the `kubectl port-forward` command running and try opening <http://localhost:8080/> in your browser. You should see the generated HTML file!

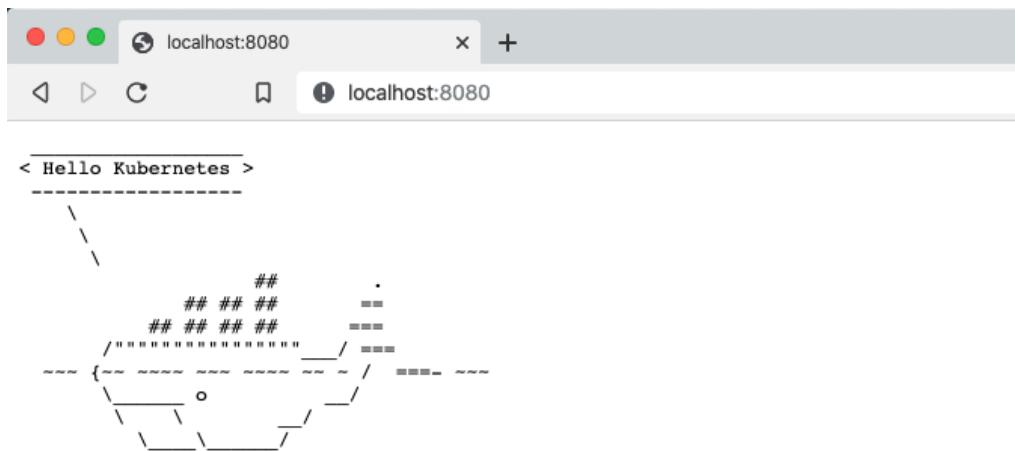


Figure 2.4 The generated HTML file content from the docker/whalesay image is an ASCII rendering of a cute whale with a speech bubble of greeting passed as a command argument. The port-forward command allows port 80 of the Pod (HTML) to be accessed on port 8080 of the localhost.

Exercise 2.1

Now that your NGINX Pod is running, use the `kubectl exec` command to get a shell on the running container. HINT: The command would be something like `kubectl exec -it`

<POD_NAME> -- /bin/bash. Poke around in the shell. Run ls, df, and ps -ef as well as other Linux commands. What happens if you terminate the nginx process?

As the final step in this exercise, let's delete the pod to free up cluster resources. The pod can be deleted using the following command:

```
$ kubectl delete pod nginx
pod "nginx" deleted
```

2.2 Declarative vs. Imperative Object Management

The Kubernetes kubectl command-line tool is used to create, update, and manage Kubernetes objects and supports imperative commands, imperative object configuration, and declarative object configuration.¹³ Let's go through a real-world example that demonstrates the difference between an imperative/procedural configuration and a declarative configuration in Kubernetes. First, let's look at how kubectl can be used imperatively.

NOTE Declarative vs. Imperative Please refer to Section 1.4.1 for a detailed explanation of Declarative vs. Imperative. In the example below, let's create a script that will deploy an nginx service with three replicas and some annotations on the deployment.

Listing 2.2 Imperative kubectl commands (`imperative-deployment.sh`).

```
#!/bin/sh
kubectl create deployment nginx-imperative --image=nginx:latest #A
kubectl scale deployment/nginx-imperative --replicas 3 #B
kubectl annotate deployment/nginx-imperative environment=prod #C
kubectl annotate deployment/nginx-imperative organization=sales #D
```

#A Create a new deployment object called nginx-imperative.

#B Scale the nginx-imperative deployment to have three replicas of the pod.

#C Add an annotation with the key environment and value prod to the nginx-imperative deployment.

#D Add an annotation with the key organization and value sales to the nginx-imperative deployment.

Try running the script against your minikube cluster and check that the deployment was created successfully.

```
$ imperative-deployment.sh
deployment.apps/nginx-imperative created
deployment.apps/nginx-imperative scaled
deployment.apps/nginx-imperative annotated
deployment.apps/nginx-imperative annotated
$ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
nginx-imperative  3/3     3           3           27s
```

Great! The deployment was created as expected. But now, let's edit our deployment.sh script to change the value of the organization annotation from sales to marketing and then rerun the script.

```
$ imperative-deployment-new.sh
```

¹³ <https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/>

```
Error from server (AlreadyExists): deployments.apps "nginx-imperative" already exists
deployment.apps/nginx-imperative scaled
error: --overwrite is false but found the following declared annotation(s): 'environment'
      already has a value (prod)
error: --overwrite is false but found the following declared annotation(s): 'organization'
      already has a value (sales)
```

As you can see, the new script failed because the deployment and annotations already exist. To make it work, we would need to enhance our script with additional commands and logic to handle the update case in addition to the creation case. Sure, this can be done, but it turns out we don't have to do all that work because `kubectl` already can examine the current state of the system and "do the right thing" using declarative object configuration.

The following manifest defines a deployment identical to the one created by our script (except the deployment's name is `nginx-declarative`).

Listing 2.3 Declarative (`declarative-deployment.yaml`).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-declarative
  annotations:
    environment: prod
    organization: sales
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```

We can use the semi-magical `kubectl apply` command to create the `nginx-declarative` deployment.

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative created
$ kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-declarative   3/3     3           3           5m29s
nginx-imperative   3/3     3           3           24m
```

After running the `apply`, we see the `nginx-declarative` deployment resource created. But what happens when we run `kubectl apply` again?

```
$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative unchanged
```

Notice the change in the output message - the second time `kubectl apply` was run, the program detected no changes needed to be made and subsequently reported that the deployment was unchanged. This is a subtle but critical difference between a `kubectl create` versus a `kubectl apply`. A `kubectl create` will fail if the resource already exists. The `kubectl apply` command first detects the resource exists and performs a create operation if the object doesn't exist or an update if it already exists.

As with the imperative example, what if we want to change the value of the organization annotation from `sales` to `marketing`? Let's edit the `declarative-deployment.yaml` file and change the `metadata.annotations.organization` field from `sales` to `marketing`. But before we run `kubectl apply` again, let's run `kubectl diff`.

```
$ kubectl diff -f declarative-deployment.yaml
:
-   organization: sales
+   organization: marketing #A
  creationTimestamp: "2019-10-15T00:57:44Z"
-   generation: 1
+   generation: 2 #B
  name: nginx-declarative
  namespace: default
  resourceVersion: "347771"

$ kubectl apply -f declarative-deployment.yaml
deployment.apps/nginx-declarative configured
```

#A The value of the organization label was changed from sales to marketing
#B The generation of this resource was changed by the system when doing kubectl apply

As you can see, `kubectl diff` correctly identified that the organization was changed from `sales` to `marketing`. We also see that `kubectl apply` successfully applied the new changes.

In this exercise, both the imperative and declarative examples result in a deployment resource configured in precisely the same way. And at first glance, the imperative approach may appear to be much simpler. It contains only a few code lines compared to the declarative deployment spec's verbosity that is five times the script's size. However, it contains problems that make it a poor choice to use in practice.

1. The code is not idempotent and may have different results if executed more than once. If RUN a second time, an error will be thrown complaining that the deployment `nginx` already exists. In contrast, the deployment spec is idempotent, meaning it can be applied as many times as needed, handling the case where the deployment already exists.
2. It is more difficult to manage changes to the resource over time, especially when the difference is subtractive. Suppose you no longer wanted "organization" to be annotated on the deployment. Simply removing the "`kubectl annotate`" command from the scripted code would not help since it would do nothing to remove the existing deployment's annotation. A separate operation would be needed to remove it. On the other hand, with the declarative approach, you only need to remove the annotation line from the spec, and Kubernetes would take care of removing the annotation to reflect your desired state.

3. It is more difficult to *understand* changes. If a team member sent a pull request modifying the script to do something differently, it would be like any other source code review. The reviewer would need to mentally walk through the script's logic to verify the algorithm achieves the desired outcome. There can even be bugs in the script. On the other hand, a pull request that changes a declarative deployment specification clearly shows the change to the system's desired state. It is simpler to review, as there is no logic to check, only a configuration change.
4. The code is not atomic, meaning that if one of the four commands in the script failed, the system's state would be partially changed and wouldn't be in the original state, nor would it be in the desired state. With the declarative approach, the entire spec is received as a single request, and the system attempts to fulfill all aspects of the desired state as a whole.

As you can imagine, what started as a simple shell script, would need to become more and more complicated to achieve idempotency. There are dozens of options available in the Kubernetes deployment spec. With the scripted approach, if/else checks would need to be littered throughout the script to understand the existing state and conditionally modify the deployment.

2.2.1 How Declarative Configuration Works

As we saw in the previous exercise, declarative configuration management is powered by the `kubectl apply` command. In contrast with imperative `kubectl` commands, like `scale` and `annotate`, the `kubectl apply` command has just one parameter: the path to the file containing the resource manifest.

```
kubectl apply -f ./resource.yaml
```

The command is responsible for figuring out which changes should be applied to the matching resource in the Kubernetes cluster and update the resource using the Kubernetes API. It is a critical feature that makes Kubernetes a perfect fit for GitOps. Let's learn more about the logic behind `kubectl apply` and understand what it can and cannot do. To understand which problems `kubectl apply` is solving, let's go through different scenarios using the Deployment resource we created above.

The simplest scenario is when the matching resource does not exist in the Kubernetes cluster. In this case, `kubectl` just creates a new resource using the manifest stored in the specified file.

If the matching resource exists, why doesn't `kubectl` just replace it? The answer is obvious if you look at the complete manifest resource using the `kubectl get` command. Below is a partial listing of the Deployment resource that was created in the example above. Some parts of the manifest have been omitted for clarity (indicated with ellipses, e.g., "...").

```
$ kubectl get deployment nginx-declarative -o=yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
```

```

deployment.kubernetes.io/revision: "1"
environment: prod
kubectl.kubernetes.io/last-applied-configuration: |
  { ... }
organization: marketing
creationTimestamp: "2019-10-15T00:57:44Z"
generation: 2
name: nginx-declarative
namespace: default
resourceVersion: "349411"
selfLink: /apis/apps/v1/namespaces/default/deployments/nginx-declarative
uid: d41cf3dc-a3e8-40dd-bc81-76af4a032b1
spec:
  progressDeadlineSeconds: 600
  replicas: 3
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx-declarative
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    ...
status:
  ...

```

As you may have noticed, a live resource manifest includes all the fields specified in the file plus dozens of new fields such as additional metadata, the `status` field, and other fields in the resource spec. All these additional fields are populated by the Deployment controller and contain important information about the resource's running state. The controller populates information about resource state in the `status` field and applies default values of all unspecified optional fields, such as `revisionHistoryLimit` and `strategy`. To preserve this information, `kubectl apply` merges the manifest from the specified file and the live resource manifest. As a result, the command updates only fields specified in the file, keeping everything else untouched. So if we decide to scale down the deployment and change the `replicas` field to 1, then `kubectl` changes only that field in the live resource and saves it back to Kubernetes using an update API.

In real life, we don't want to control all possible fields which influence resource behavior in a declarative way. It makes sense to leave some room for imperativeness and skip fields that should be changed dynamically. The `replicas` field of the Deployment resource is a perfect example. Instead of hard-coding the number of replicas you want to use, the Horizontal Pod Autoscaler can be used to dynamically scale up or scale down your application based on load.

NOTE *Horizontal Pod Autoscaler* The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, or replica set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metric).

Let's go ahead and remove the `replicas` field from the Deployment manifest. After applying this change, the `replicas` field is reset to the default value of one replica. But wait! The `kubectl apply` command updates only those fields which are specified in the file and ignores the rest. How does it know that the `replicas` field was deleted? The additional information which allows `kubectl` to handle the delete use case is hidden in an annotation of the live resource. Every time the `kubectl apply` command updates a resource, it saves the input manifest in the `kubectl.kubernetes.io/last-applied-configuration` annotation. So when the command is executed the next time, it retrieves the most recently applied manifest from the annotation, representing the common ancestor of the new desired manifest and live resource manifest. This allows `kubectl` to execute a three-way diff/merge and properly handle the case where some fields are removed from the resource manifest.

NOTE Three-Way Merge A three-way merge is a merge algorithm that automatically analyzes differences between two files while also considering the origin or the common ancestor of both files.

Finally, let's discuss the situations where `kubectl apply` might not work as expected and should be used carefully.

First off, you typically should not mix imperative commands, such as `kubectl edit` or `kubectl scale`, with declarative resource management. This will make the current state not match the `last-applied-configuration` annotation and will defeat the merge algorithm `kubectl` uses to determine deleted fields. The typical scenario is when you experiment with the resource using `kubectl edit` and want to rollback changes by applying the original manifests stored in files. Unfortunately, it might not work since changes made by the `kubectl edit` command are not stored anywhere. For example, if you temporarily add the `resource limits` field to the deployment, the `kubectl apply` won't remove it since the `limits` field is not mentioned in the `last-applied-configuration` annotation or the manifest from the file. The `kubectl replace` command similarly ignores the `last-applied-configuration` annotation and removes that annotation altogether after applying the changes. So if you make any changes imperatively, you should be ready to undo the changes using imperative commands before continuing with declarative configuration.

You should also be careful when you want to stop managing fields declaratively. A typical example of this problem is adding the Horizontal Pod Autoscaler to manage scaling the number of replicas for an existing deployment. Typically, before introducing Horizontal Pod Autoscaler, the number of deployment `replicas` is managed declaratively. To pass control of the `replicas` field over to the Horizontal Pod Autoscaler, the `replicas` field must first be deleted from the file which contains the Deployment manifest. This is so the next `kubectl apply` does not override the `replicas` value set by the Horizontal Pod Autoscaler. However, don't forget that the `replicas` field might also be stored in the `last-applied-configuration` annotation. If that is the case, the missing `replicas` field in the manifest file will be treated as a field deletion, so whenever `kubectl apply` is run, the `replicas` value set imperatively by HPA will be removed from the live Deployment. The Deployment will scale down to the default of one replica.

In this section, we covered the different mechanisms for managing Kubernetes objects: imperative and declarative. You also learned a little about the internals of kubectl and how it identifies changes to apply to live objects. But at this point, you may be wondering what all this has to do with GitOps. The answer is simple: everything! Understanding how kubectl and Kubernetes manages changes to live objects is critical for understanding how the GitOps tools discussed in later chapters identify if the Git repository holding the Kubernetes configuration is in sync with the live state and how it tracks and applies changes.

2.3 Controller Architecture

So far, we've learned about Kubernetes' declarative nature and the benefits it provides. Let's talk about what is behind each Kubernetes resource; the controller architecture. Understanding how controllers work will help us use Kubernetes more efficiently and understand how it can be extended.

Controllers are "brains" that understand what a particular kind of resource manifest means and execute the necessary work to make the system's actual state match the desired state as described by the manifest. Each controller is typically responsible for only one resource type. Through listening to the API server events related to the resource type being managed, the controller continuously watches for changes to the resource's configuration and performs the necessary work to move the current state towards the desired state. An essential feature of Kubernetes controllers is the ability to delegate work to other controllers. This layered architecture is very powerful and allows you to reuse functionality provided by different resource types effectively. Let's consider a concrete example to understand the delegation concept better.

2.3.1 Controller Delegation

The Deployment, ReplicaSet, and Pod resources perfectly demonstrate how delegation empowers Kubernetes.

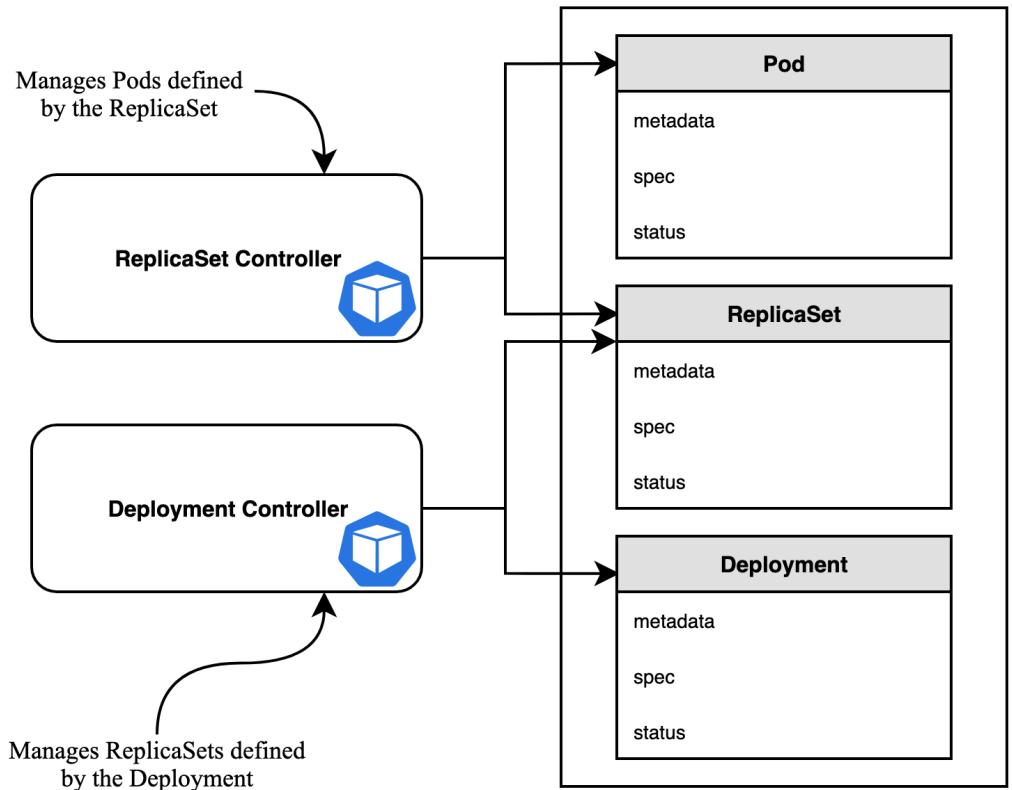


Figure 2.5 Kubernetes allow for a resource hierarchy. Higher-level resources providing additional functionality, such as ReplicaSets and Deployments, can manage other higher-level resources or primary resources, such as Pods. This is implemented through a series of controllers, each managing events related to the resources it controls.

The Pod provides the ability to run one or more containers that have requested resources on a node in the cluster. This allows the Pod controller to focus simply on running an instance of an application and abstracts the logic related to infrastructure provisioning, scaling up and down, networking, and other complicated details, leaving those to other controllers. Although the Pod resource provides many features, it is still not enough to run an application in production. We need to run multiple instances of the same application (for resiliency and performance), which means we need multiple Pods. The ReplicaSet controller solves this problem. Instead of directly managing multiple containers, it orchestrates multiple Pods and delegates the container orchestration to the Pod resource. Similarly, the Deployment controller leverages functionality provided by ReplicaSets to implement various deployment strategies such as rolling updates.

CONTROLLER DELEGATION BENEFIT With controller delegation, Kubernetes functionality can be easily extended to support new capabilities. For example, services that are not backward compatible can only be deployed with a Blue/Green strategy (not rolling-update). Controller delegation allows a new controller to be rewritten to support blue/green deployment and still leverage the Deployment Controller functionality through delegation without re-implementing the Deployment Controller's core functionality.

So, as you can see from this example, controller delegation allows Kubernetes to build progressively more complex resources from more simple ones.

2.3.2 Controller Pattern

Although all controllers have different responsibilities, the implementation of each controller follows the same simple pattern. Each controller runs an infinite loop, and every iteration reconciles the desired and the actual state of the cluster resources it is responsible for. During reconciliation, the controller is looking for differences between the actual and desired state and making the changes necessary to move the current state towards the desired state.

The desired state is represented by the spec field of the resource manifest. The question is, how does the controller know about the actual state? This information is available in the status field. After every successful reconciliation, the controller updates the status field. The status field provides information about cluster state to end-users and enables the work of higher-level controllers. Figure 2.6 demonstrates the reconciliation loop.

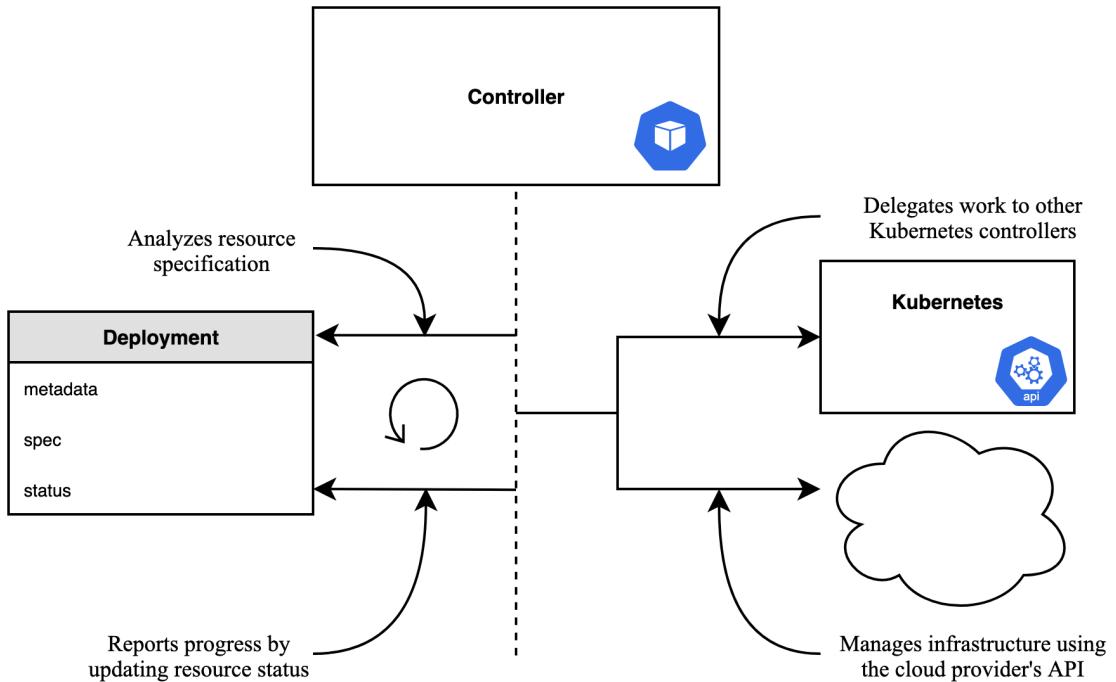


Figure 2.6 A Controller operates in a continuous reconciliation loop where it attempts to converge the desired state as defined in the spec with the current state. Changes and updates to the resource are reported by updating the resource status. The controller may delegate work to other Kubernetes controllers or perform other operations, such as manage external resources using the cloud provider's API.

CONTROLLERS VS. OPERATORS

Two terms that are often confused are the term operator vs. controller. In this book, the term *GitOps Operator* is used to describe continuous delivery tools instead of *GitOps Controller*. The reason for this is we are representing a specific type of controller that is application and domain-specific.

NOTE *Kubernetes Operators* A Kubernetes Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the primary Kubernetes resource and controller concepts and includes domain or application-specific knowledge to automate everyday tasks.

The terms operators and controllers are often confused since operator and controller are sometimes used interchangeably, and the line between the two is often blurred. However, another way to think about it is that the term “operator” is used to describe application-specific controllers. All operators use the controller pattern, but not all controllers are operators. Generally speaking, controllers tend to manage lower level, reusable building-

block resources, whereas operators operate at a higher level and are application-specific. Some examples of controllers are all of the built-in controllers which manage Kubernetes native types (Deployments, Jobs, Ingresses, etc.), as well as third party controllers such as cert-manager (which provisions and manages TLS certificates), and the Argo Workflow Controller, which introduces a new job-like Workflow resource in the cluster. An example of an operator is the Prometheus Operator, which manages Prometheus database installations.

2.3.3 NGINX Operator

After learning about the controller fundamentals and the differences between Controllers and Operators, we are ready to implement an Operator! The sample operator will solve a real-life task: manage a suite of NGINX servers with pre-configured static content. The operator will allow the user to specify a list of NGINX servers and configure static files mounted on each server. The task is not trivial and demonstrates the flexibility and power of Kubernetes.

DESIGN

As mentioned earlier in this chapter, Kubernetes' architecture allows you to leverage an existing controller's functionality through delegation. Our NGINX controller is going to leverage Deployment resources to delegate the NGINX deployment task.

The next question is which resource should be used to configure the list of servers and customized static content. The most appropriate existing resource is ConfigMap. According to the official Kubernetes documentation, the ConfigMap is "an API object used to store non-confidential data in key-value pairs."¹⁴ The ConfigMap can be consumed as environment variables, command-line arguments, or config files in a volume. The controller will create a Deployment for each ConfigMap and mount the ConfigMap data into the default NGINX static website directory.

¹⁴ <https://kubernetes.io/docs/reference/glossary/?core-object=true#term-configmap>

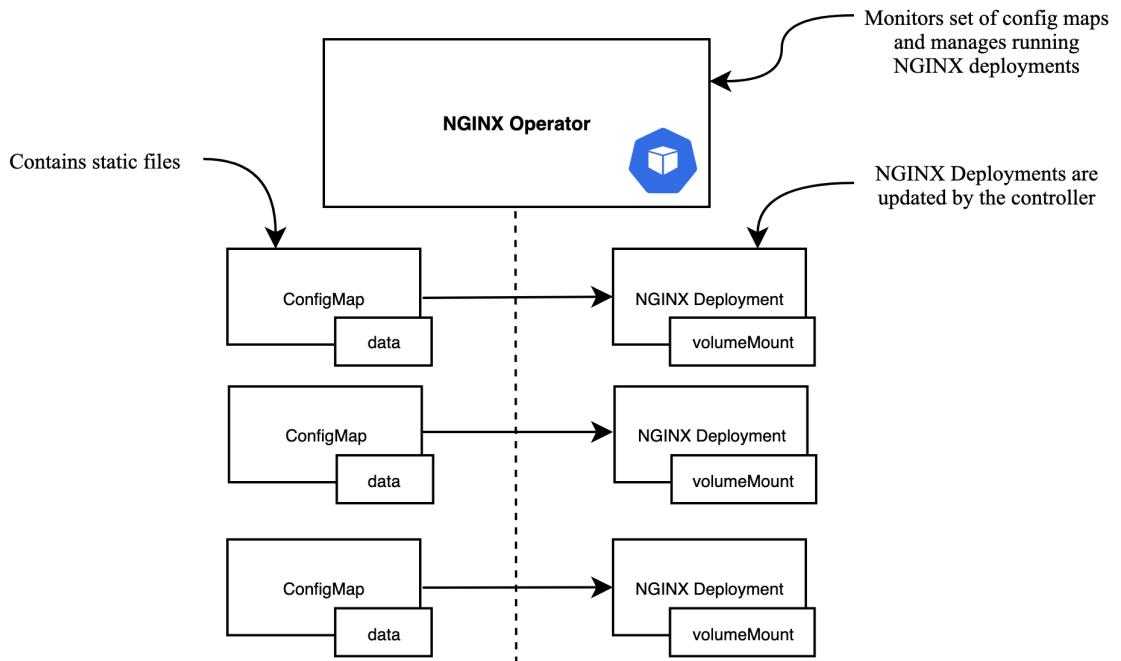


Figure 2.7 In the NGINX operator design, a ConfigMap is created containing the data to be served by NGINX. The NGINX operator creates a Deployment for each ConfigMap. Additional NGINX deployments can be created simply by creating a ConfigMap with the web page data.

IMPLEMENTATION

Once we've decided on the design of the main building blocks, it is time to write some code. Most Kubernetes related projects, including Kubernetes itself, are implemented using Golang. However, Kubernetes controllers can be implemented using any language, including Java, C++, or even Javascript. For the sake of simplicity, we are going to use a language that is most likely familiar to you: the Bash scripting language.

In the Controller Pattern section, we mentioned that each controller maintains an infinite loop and continuously reconciles the desired and actual state. In our example, the desired state is represented by the list of ConfigMaps. This most efficient way to loop through every ConfigMap change is using the Kubernetes `watch` API. The `watch` feature is provided by the Kubernetes API for most resource types and allows the caller to be notified when a resource is created, modified, or deleted. The `kubectl` utility allows watching for resource changes using the `get` command with the `--watch` flag. The `--output-watch-events` instructs `kubectl` to output the change type that might take one of the following values: `ADDED`, `MODIFIED`, or `DELETED`.

NOTE *Kubectl* version Ensure you are using the latest version of kubectl for this tutorial (or at least version 1.16 or greater. The `--output-watch-events` option was added relatively recently.

Listing 2.4 Sample ConfigMap (`sample.yaml`).

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample
data:
  index.html: hello world
```

In one window, run the following command:

```
$ kubectl get --watch --output-watch-events configmap
```

In another terminal window, run `kubectl apply -f sample.yaml` to create the sample ConfigMap. Notice the new output in the window running the `kubectl --watch` command. Now run `kubectl delete -f sample.yaml`. You should now see a `DELETED` event appear.

```
$ kubectl get --watch --output-watch-events configmap
EVENT      NAME      DATA      AGE
ADDED     sample    1        3m30s
DELETED   sample    1        3m40s
```

After running this experiment manually, you should be able to see how we can write our NGINX operator as a bash script.

The `kubectl get --watch` command outputs a new line every time a ConfigMap resource is created, changed, or deleted. The script will consume the output of `kubectl get --watch` and either create a new Deployment or delete a Deployment depending on the output ConfigMap event type. Without further delay, the full operator implementation in the code listing below.

Listing 2.5 NGINX Controller (`controller.sh`).

```
#!/usr/bin/env bash

kubectl get --watch --output-watch-events configmap \
-o=custom-columns=type:type,name:object.metadata.name \
--no-headers | \
while read next; do

  NAME=$(echo $next | cut -d' ' -f2)          #A
  EVENT=$(echo $next | cut -d' ' -f1)           #B

  case $EVENT in
    ADDED|MODIFIED)
      kubectl apply -f - << EOF
apiVersion: apps/v1
kind: Deployment
metadata: { name: $NAME }
spec:
  selector:
    matchLabels: { app: $NAME }
```

```

template:
  metadata:
    labels: { app: $NAME }
    annotations: { kubectl.kubernetes.io/restartedAt: $(date) }
  spec:
    containers:
      - image: nginx:1.7.9
        name: $NAME
        ports:
          - containerPort: 80
    volumeMounts:
      - { name: data, mountPath: /usr/share/nginx/html }
  volumes:
    - name: data
  configMap:
    name: $NAME
EOF
      ;;
  DELETED)
    kubectl delete deploy $NAME
    ;;
esac
done

```

#A This `kubectl` command outputs all the events that occur for configmap objects

#B The output from `kubectl` is processed by this infinite loop

#C The name of the configmap and the event type is parsed from the `kubectl` output

#D If the configmap has been ADDED or MODIFIED, apply the NGINX deployment manifest (everything between the two EOF tags) for that configmap

#E If the configmap has been DELETED, delete the NGINX deployment for that configmap

TESTING

Now that the implementation is done, we are ready to test our controller. In real life, the controller is packaged into a Docker image and runs inside the cluster. It is OK to run the controller outside of the cluster for testing purposes, which is precisely what we are going to do. Using instructions from Appendix A, start a minikube cluster, save the controller code into a file called `controller.sh` and start it using the `bash` command below.

Note: This example requires `kubectl` version 1.16 or later.

```
$ bash controller.sh
```

The controller is running and waiting for the ConfigMap. Let's create one. Refer to listing 2.4 for the manifest of the ConfigMap.

We create the ConfigMap using the `kubectl apply` command.

```
$ kubectl apply -f sample.yaml
configmap/sample created
```

The controller notices the change and creates an instance of Deployment using the `kubectl apply` command.

```
$ bash controller.sh
deployment.apps/sample created
```

Exercise 2.2

Try accessing the NGINX by forwarding port 80 locally to make sure the controller works as expected. Try to delete or modify the ConfigMap and see how the controller reacts accordingly.

Exercise 2.3

Create additional ConfigMaps to launch an NGINX server for each member of your family that displays "Hello <name>!". Also, don't forget to call/text/SnapChat them IRL.

Exercise 2.4

Write a Dockerfile to package the NGINX controller. Deploy it to your test Kubernetes cluster. Hint: You will need to create RBAC resources for the operator.

2.4 Kubernetes + GitOps

GitOps assumes that every piece of infrastructure is represented as a file stored in a revision control system, and there is an automated process that seamlessly applies changes to the application runtime environment. Without a system like Kubernetes, this is, unfortunately, easier said than done. There are just too many things to worry about and many different technologies that do not work well together. These two assumptions often become an unsolvable obstacle that prevents the implementation of an efficient infrastructure-as-code process.

Kubernetes has dramatically improved the situation. As Kubernetes gained more and more adoption, the idea of infrastructure-as-code has evolved, which resulted in the creation of new tooling which implements GitOps. So what is so special about Kubernetes, and how and why did it lead to the rise of GitOps?

Kubernetes enables GitOps by (1) fully embracing declarative APIs as its primary mode of operation and (2) providing the controller patterns and backend framework necessary to implement those APIs. The system was designed with the principles of declarative specifications and eventual consistency and convergence from its inception.

NOTE *Eventual consistency* Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

This decision is what led to the prominence of GitOps in Kubernetes. Unlike traditional systems, in Kubernetes there are almost no APIs that can modify just a subset of some existing resources. For example, there is no API (and never will be) that changes only the container image of a Pod. Instead, the Kubernetes API server expects all API requests to provide a complete manifest of the resource to the API server. It was an intentional decision not to give any "convenience" APIs to users. As a result, Kubernetes users are essentially forced into a declarative mode of operation, which leads these same users into the need to store these declarative specifications somewhere. Git became the natural medium to store these specifications, and GitOps then became the natural delivery tool to deploy these manifests from Git.

2.5 Getting Started with CI/CD

Now that you've learned the basic architecture and principles of a Kubernetes controller and how Kubernetes is a good fit for GitOps, it's time to implement your own GitOps operator. In this tutorial, we will first be creating a rudimentary GitOps operator to drive continuous delivery. This is followed by an example of how you would integrate continuous integration (CI) with a GitOps-based continuous delivery (CD) solution.

2.5.1 Basic GitOps Operator

To implement your own GitOps operator, a continuously running control loop needs to be implemented which performs the following three steps:

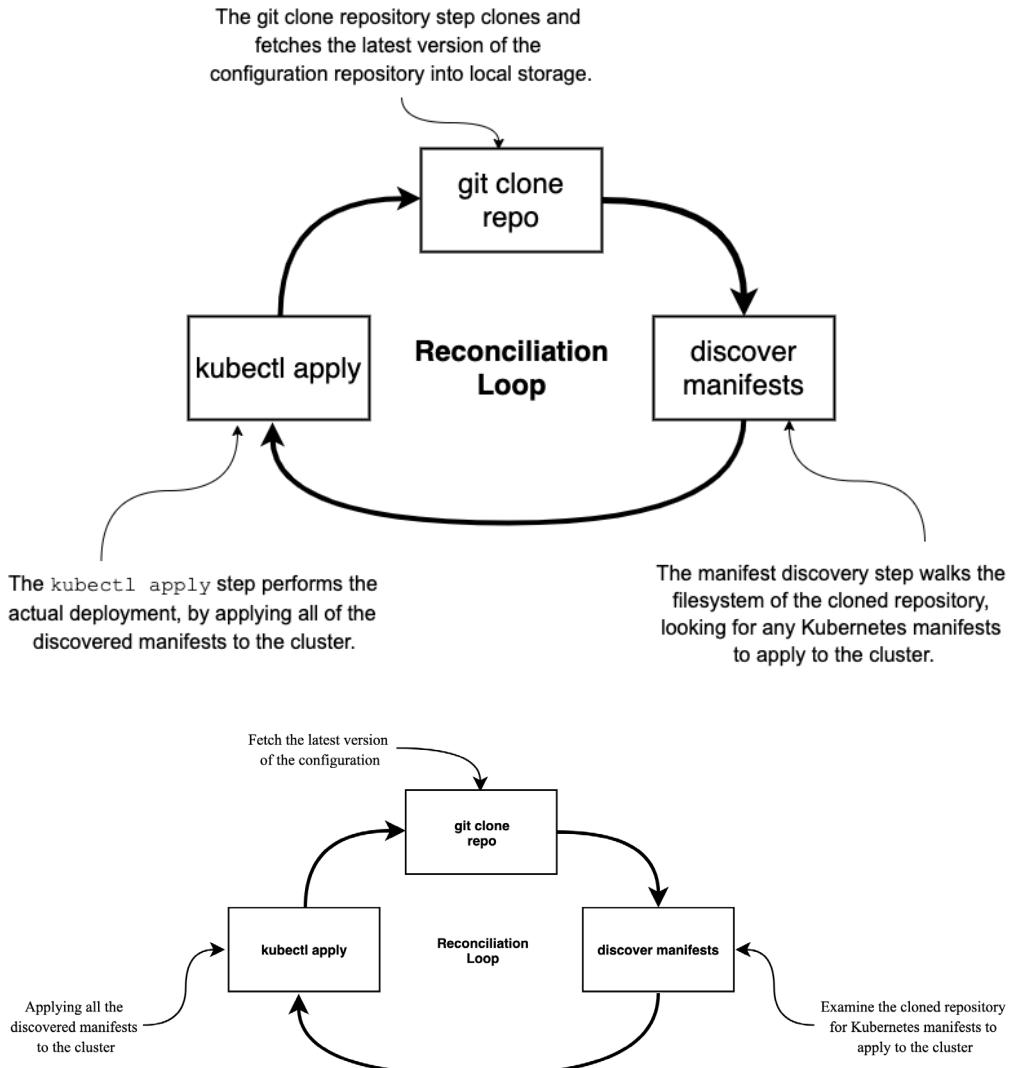


Figure 2.8 The GitOps reconciliation loop begins by cloning the repository to fetch the configuration repository's latest version into local storage. Next, the manifest discovery step walks the cloned repository's filesystem, looking for any Kubernetes manifests to apply to the cluster. Lastly, the kubectl apply step performs the actual deployment by applying all of the discovered manifests to the cluster.

While this control loop could be implemented in any number of ways, most simply, it could be implemented as a Kubernetes CronJob.

Listing 2.6 CronJob GitOps Operator (`gitops-cronjob.yaml`).

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: gitops-cron
  namespace: gitops
spec:
  schedule: "*/5 * * * *"          #A
  concurrencyPolicy: Forbid       #B
  jobTemplate:
    spec:
      backoffLimit: 0             #D
      template:
        spec:
          restartPolicy: Never   #E
          serviceAccountName: gitops-serviceaccount #F
      containers:
        - name: gitops-operator
          image: gitopsbook/example-operator:v1.0 #G
          command: [sh, -e, -c]                  #H
          args:
            - git clone https://github.com/gitopsbook/sample-app-deployment.git
/ttmp/example &&
          find /tmp/example -name '*.yaml' -exec kubectl apply -f {} \;

```

#A Execute the GitOps reconciliation loop every 5 minutes

#B Prevent concurrent executions of the job

#C Don't retry failed jobs since this is a recurring CronJob; retries happen naturally

#D Don't restart the container when it completes

#E A Kubernetes service account that has sufficient privileges to create and modify objects into the cluster

#F The docker image that has the git, find, and kubectl binaries preloaded into it

#G The command and args fields contain the actual logic of the GitOps reconciliation loop.

The job template spec contains the meat of the operator logic. The CronJob “gitops-cron” contains the control loop logic that deploys manifests from Git to the cluster on a regularly scheduled basis. The `schedule` field is a cron expression, which in this example will result in the job being executed every 5 minutes. Setting the `concurrencyPolicy` to `Forbid` prevents concurrent executions of the job, allowing the current execution to complete before attempting to start a second. Note that this would only happen if a single execution takes longer than 5 minutes.

The `jobTemplate` is a Kubernetes job template spec. The job template spec contains a pod template spec (`jobTemplate.spec.template.spec`), which is the same spec that you may be familiar with when writing Kubernetes manifests for Deployments, Pods, Jobs, etc. The `backoffLimit` specifies the number of retries before considering a Job as failed. A value of zero means that it will not retry. Since this is a recurring CronJob, retries happen naturally, so there is no need to retry immediately. A `restartPolicy` of `Never` is required to prevent the job from restarting the container when it completes, which is a container's normal behavior. The `serviceAccountName` field references a Kubernetes service account with sufficient privileges to create and modify objects in the cluster. Since this operator could potentially deploy any type of resource, the `gitops-operator` service account should be bound to an admin-level ClusterRole.

The command and args fields contain the actual logic of the GitOps reconciliation loop. It is comprised of only two commands:

- git clone - which clones the latest repository to local storage
- find - which discovers YAML files in the repo, and for each YAML file located, executes the `kubectl apply` command against each file.

To use this, simply apply the CronJob to the cluster. Note that you would first need to apply the following supporting resources:

Listing 2.7 CronJob GitOps Resources (`gitops-resources.yaml`).

```

apiVersion: v1                                     #A
kind: Namespace
metadata:
  name: gitops

---
apiVersion: v1                                     #B
kind: ServiceAccount
metadata:
  name: gitops-serviceaccount
  namespace: gitops

---
apiVersion: rbac.authorization.k8s.io/v1           #C
kind: ClusterRoleBinding
metadata:
  name: gitops-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: ServiceAccount
  name: gitops-serviceaccount
  namespace: gitops

```

#A Namespace `gitops` is where the CronJob and ServiceAccount will live.

#B ServiceAccount `gitops-serviceaccount` is the Kubernetes service account that will have privileges to deploy to the cluster.

#C ClusterRoleBinding `gitops-operator` binds/grants cluster admin-level privileges to the ServiceAccount, `gitops-serviceaccount`.

NOTE Multi-resource YAML files Management of multiple resources can be simplified by grouping them in the same file (separated by --- in YAML). The above listing is an example of a single YAML file defining multiple related resources.

This example is very primitive, meant to illustrate the fundamental concepts of a GitOps continuous delivery operator. It is not meant for any real production use since it lacks many features needed in a real-world production environment. For example, it cannot “prune” any resources that are no longer defined in Git. Another limitation is that it does not deal with any credentials required to connect to the Git repository.

Exercise 2.5

Modify the CronJob to point to your own GitHub repository. Apply the new CronJob and add YAML files to your repository. Verify the corresponding Kubernetes resources are created.

2.5.2 Continuous Integration Pipeline

In the previous section, we've implemented a basic GitOps continuous delivery (CD) mechanism which continuously delivers manifests in a Git repository to the cluster. The next step is to integrate this process with a continuous integration (CI) pipeline, which publishes new container images and updates the Kubernetes manifests with the new image. GitOps integrates well with any CI system as the process is more or less the same as a typical build pipeline. The main difference is that instead of the CI pipeline communicating directly to the Kubernetes API server, it instead commits the desired change into Git and trusts that sometime later, the new changes will be detected by the GitOps operator and applied.

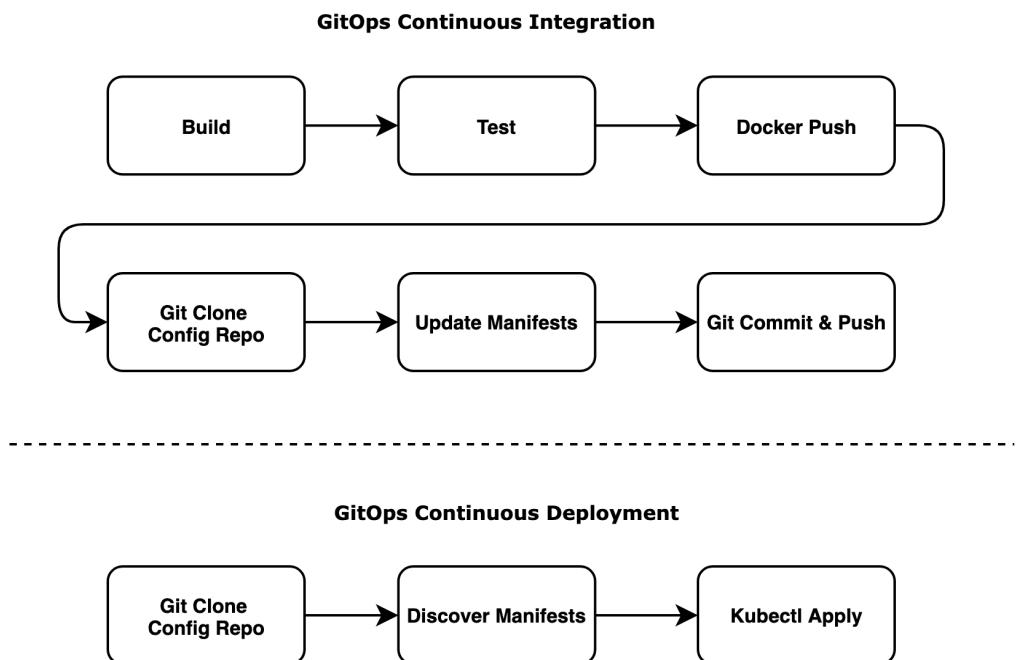


Figure 2.9 A GitOps Continuous Integration (CI) pipeline is similar to a typical CI pipeline. The code is built and tested, and then the artifact (a tagged Docker image) is pushed to the image registry. The additional step is the GitOps CI pipeline also updates the manifests in the configuration repo with the latest image tag. This update may trigger a GitOps Continuous Deployment job to apply the updated manifests to the cluster.

The goal of a GitOps CI pipeline is to:

1. Build your application and run unit testing as necessary

2. Publish a new container image to a container registry
3. Update the Kubernetes manifests in Git to reflect the new image

The following example is a typical series of commands that would be executed in a CI pipeline to achieve this.

Listing 2.8 Example GitOps CI ([gitops-ci.sh](#)).

```
export VERSION=$(git rev-parse HEAD | cut -c1-7)          #A
make build
make test

export NEW_IMAGE="gitopsbook/sample-app:${VERSION}"        #C
docker build -t ${NEW_IMAGE} .
docker push ${NEW_IMAGE}

git clone http://github.com/gitopsbook/sample-app-deployment.git  #D
cd sample-app-deployment
kubectl patch \
  --local \
  -o yaml \
  -f deployment.yaml \
  -p "spec:
    template:
      spec:
        containers:
          - name: sample-app
            image: ${NEW_IMAGE}" \
> /tmp/newdeployment.yaml
mv /tmp/newdeployment.yaml deployment.yaml
#E
git commit deployment.yaml -m "Update sample-app image to ${NEW_IMAGE}"
git push                                #F
```

#A Use the first seven characters of the current commit-SHA as the version to uniquely identify the artifacts from this build.

#B Build and test your application's binaries as you usually would.

#C Build the container image and push it to a container registry. Incorporate the unique version as part of the container image tag.

#D Clone the Git deployment repo containing the Kubernetes manifests

#E Update the manifests with the new image.

#F Commit and push the manifest changes to the deployment configuration repo.

This example pipeline is just one way that a GitOps CI pipeline may look. There are some important points to highlight regarding the different choices you might make that would better suit your needs.

IMAGE TAGS AND THE TRAP OF THE “LATEST” TAG

Notice in the first two steps of the example pipeline, the current Git commit-SHA of the application's Git repository is used as a version variable, which is then incorporated as part of the container's image tag. A resulting container image in the example pipeline might look like `gitopsbook/sample-app:cc52a36`, where `cc52a36` is the commit-SHA at the time of the build.

It is important to use a unique version string (like a commit-SHA) that is different in each build since the version is incorporated as part of the container image tag. A common mistake that people make is to use `latest` as their image tag (e.g., `gitopsbook/sample-app:latest`) or reusing the same image tag from build to build. A naive pipeline might make the following mistake:

```
make build
docker build -t gitopsbook/sample-app:latest .
docker push gitopsbook/sample-app:latest
```

Reusing image tags from build to build is a terrible practice for several reasons:

The first reason why container tags should not be reused is that when container image tags are reused, Kubernetes will not deploy the new version to the cluster. This is because the second time the manifests are attempted to be applied, Kubernetes would not detect any change in the manifests, and the second `kubectl apply` would have zero effect. For example, say build #1 publishes the image `gitopsbook/sample-app:latest` and deploys it to the cluster. The Deployment manifest for this might look something like:

Listing 2.9 Sample App Deployment ([deployment.yaml](#)).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - image: gitopsbook/sample-app:latest
          name: sample-app
          command:
            - /app/sample-app
          ports:
            - containerPort: 8080
```

When build #2 runs, even though a new container image for `gitopsbook/sample-app:latest` has been pushed to the container registry, the Kubernetes Deployment YAML for the application is the same as it was in build #1. The Deployment specs are the same from the perspective of Kubernetes; there is no difference between what is being applied in build #1 versus build #2. Kubernetes treats the second `apply` as a no-op (no operation) and does nothing. For Kubernetes to redeploy, something needs to be different in the Deployment spec from the first build to the second. Using unique container image tags ensures there is a difference.

Another reason for incorporating a unique version into the image tag is because it enables traceability. By incorporating something like the application's Git commit-SHA into the tag, there is never any question about what version of the software is currently running in the cluster. For example, you could run the following kubectl command, which outputs the images of all deployments in the namespace:

```
$ kubectl get deploy -o wide | awk '{print $1,$7}' | column -t
NAME      IMAGES
sample-app gitopsbook/sample-app:508d3df
```

By using the convention of tying container image tags to Git commit-SHAs of your application repository, you can trace the currently running version of the sample-app to commit 508d3df. From there, you have full knowledge of exactly what version of your application is running in the cluster.

The third and possibly most important reason for not reusing image tags such as `latest` is that rollback to the older version becomes impossible. When you reuse image tags, you are overriding or rewriting the meaning of that overwritten image. Imagine the following sequence of events:

1. Build #1 publishes the container image `gitopsbook/sample-app:latest` and deploys it to the cluster.
2. Build #2 re-publishes the container image `gitopsbook/sample-app:latest`, overwriting the image tag deployed in build #1. It redeploys this image to the cluster.
3. Sometime after build #2 is deployed, it is discovered that a severe bug exists in the latest version of the code, and immediate rollback is necessary to the version created in build #1.

There is no easy way to redeploy the version of the sample-app created during build #1 because there is no image tag representing that version of the software. The second build overwrote the `latest` image tag, effectively making the original image unreachable (at least not without extreme measures).

For these reasons, it is discouraged to reuse image tags, such as `latest`, at least in production environments. With that said, in dev and test environments, continuously creating new and unique image tags (which likely never get cleaned up) could cause an excessive amount of disk usage to be used in your container registry or become unmanageable just by the sheer number of image tags. In these scenarios, reusing image tags may be appropriate, understanding Kubernetes' behavior of not doing anything when the same specification is applied twice.

NOTE `kubectl rollout restart` Kubectl has a convenience command, `kubectl rollout restart`, which causes all the pods of a deployment to restart (even if the image tag is the same). It is useful in dev and test scenarios where the image tag has been overwritten, and redeploy is desired. It works by injecting an arbitrary timestamp into the pod template metadata annotations. This causes the pod spec to be different from what it was before, which causes a regular, rolling update of the pods.

One thing to note is that our CI example uses a Git commit-SHA as the unique image tag. But instead of a Git commit-SHA, the image tag could incorporate any other unique identifier, such as a semantic version, a build number, a date/time string, or even a combination of these pieces of information.

NOTE *Semantic Version* A semantic version is a versioning methodology that uses a three-digit convention (MAJOR.MINOR.PATCH) to convey the meaning of a version (e.g., v2.0.1). MAJOR is incremented when there are incompatible API changes. MINOR is incremented when functionality is added in a backward-compatible manner. PATCH is incremented when there are backward-compatible bug fixes.

2.6 Summary

- Kubernetes is a container orchestration system for deployment, scaling, and management of containers.
- Basic Kubernetes objects are Pod, Service, and Volume.
- Kubernetes Control Plane consists of a kube-apiserver, kube-controller-manager, and kube-scheduler.
- Each Kubernetes worker node runs kubelet and kube-proxy.
- How to deploy and access pods in Kubernetes.
- How to deploy using Imperative and Declarative syntax
- Controllers are the “brains” in Kubernetes to bring the running state into the desired state.
- How to implement a Kubernetes Operator
- Kubernetes configuration is declarative
- GitOps complements Kubernetes due to its declarative nature
- GitOps operators trigger deployments to your Kubernetes cluster based on changes to revision-controlled configuration files stored in Git.
- How to implement a GitOps operator
- How to create a Continuous Integration pipeline

3

Environment Management

This chapter covers:

- What is an environment?
- Designing the “right for me” environment using namespaces
- Organizing your Git repo/branching strategy to support your environment
- Implementing Config Management for your environment

In chapter 2, you learned how GitOps can deploy applications to a runtime environment. This chapter teaches us more about those different runtime environments and how Kubernetes namespaces can define environment boundaries. We also learn about several configuration management tools (Helm, Kustomize, and Jsonnet) and how they can help manage an application's configuration consistently in multiple environments.

We recommend you read chapters 1 and 2 before reading this chapter.

3.1 Introduction to Environment Management

In software deployment, an *environment* is where code is deployed and executed. Different environments serve different purposes in the life cycle of software development. For example, a local development environment (aka laptop) is where engineers can create, test, and debug new code versions. After engineers complete the code development, the next step is to commit the changes to Git and kick off a deployment to different environments for integration testing and eventual production release. This process is known as *Continuous Integration/Continuous Deployment* (CI/CD) that typically consists of the following environments: *QA, E2E, Stage, and Prod*.

The QA environment is where the new code will be tested against hardware, data, and other production-like dependencies to ensure your service's correctness. If all tests pass in QA, the new code will be promoted to the E2E environment as a stable environment for other pre-release services to test/integrate with. QA and E2E environments are also known as pre-

production (pre-prod) environments because they do not host production traffic or use production data.

When a new version of code is ready for production release, the code will typically deploy first in the stage environment (which has access to actual production dependencies) to ensure all production dependencies are in place before the code goes live in the prod environment. For example, new code may require a new DB schema update, and the stage environment can be used to verify the new schema is in place. Configuration is done to only direct test traffic to the stage environment so that any problems introduced by the new code would not impact actual customers. However, the stage environment is typically configured to use the “real” production database operations. Tests performed on the Stage environment must be carefully reviewed to ensure they are safe to perform in production. Once all tests pass in Stage, the new code will finally be deployed in Prod for live production traffic. Since Stage and Prod both have access to production data, they are both considered production environments.

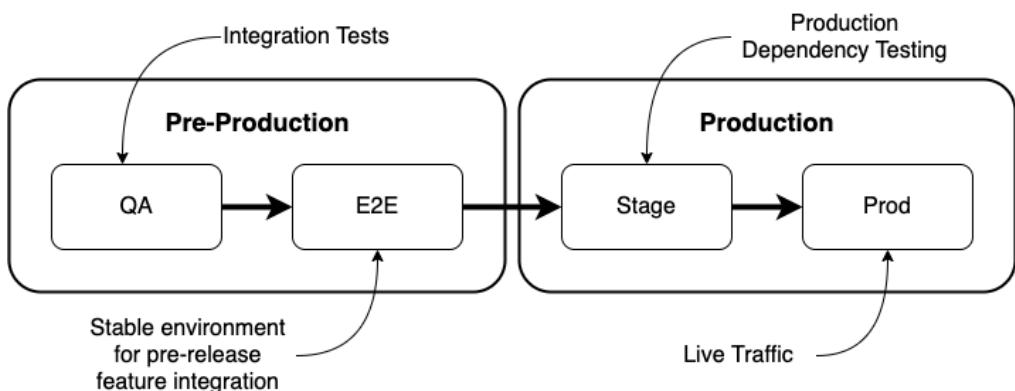


Figure 3.1 Pre-production will have a QA environment for integration testing and an E2E environment for pre-release feature integration. Production environments may have a Staging environment for production dependency testing and the actual production environment for live traffic.

3.1.1 Components of an environment

An environment is composed of three equally important components.

- Code
- Runtime Prerequisites
- Configuration

Code is the machine instructions of the application to execute specific tasks. To execute the code, runtime dependencies may also be required to execute the code. For example, Node.js code will require the Node.js binary and other npm packages in order to execute successfully. In the case of Kubernetes, all runtime dependencies and code are packaged as a deployable unit (aka Docker image) and orchestrated through the Docker Daemon. The

application's Docker image can be confidently run in any environment, from the developer's laptop to the production cluster running in the cloud, because the image encapsulates the code and all the dependencies, eliminating potential incompatibilities between environments.

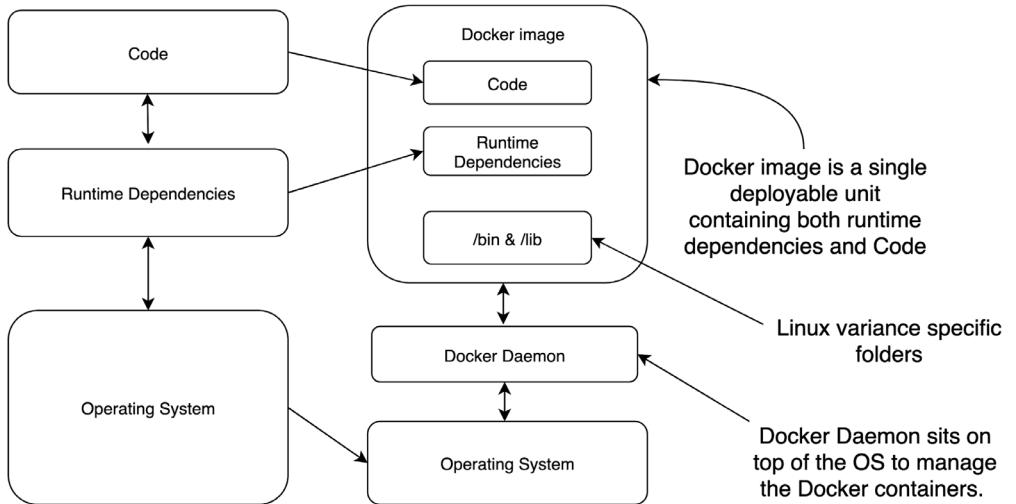


Figure 3.2 The left side represents a non-container based deployment which requires Operating Systems and Runtime Dependencies before the code can be deployed. The right side represents a container-based deployment that contains both code and runtime dependencies.

Configuration of environment-specific application properties are typically deployed along with code and runtime dependencies, so the "application instance" can behave and connect to the correct dependencies per environment. Each environment could contain DB storage, Distributed Cache, or Messaging (i.e., "data") for isolation. Environments also have their own networking policy for ingress and egress for traffic isolation and custom access control. For example, ingress and egress can be configured to block traffic between pre-prod and prod environments for security. Access Control can be configured to restrict access to the production environment to only a small set of engineers, while pre-prod environments are accessible by the entire development team.

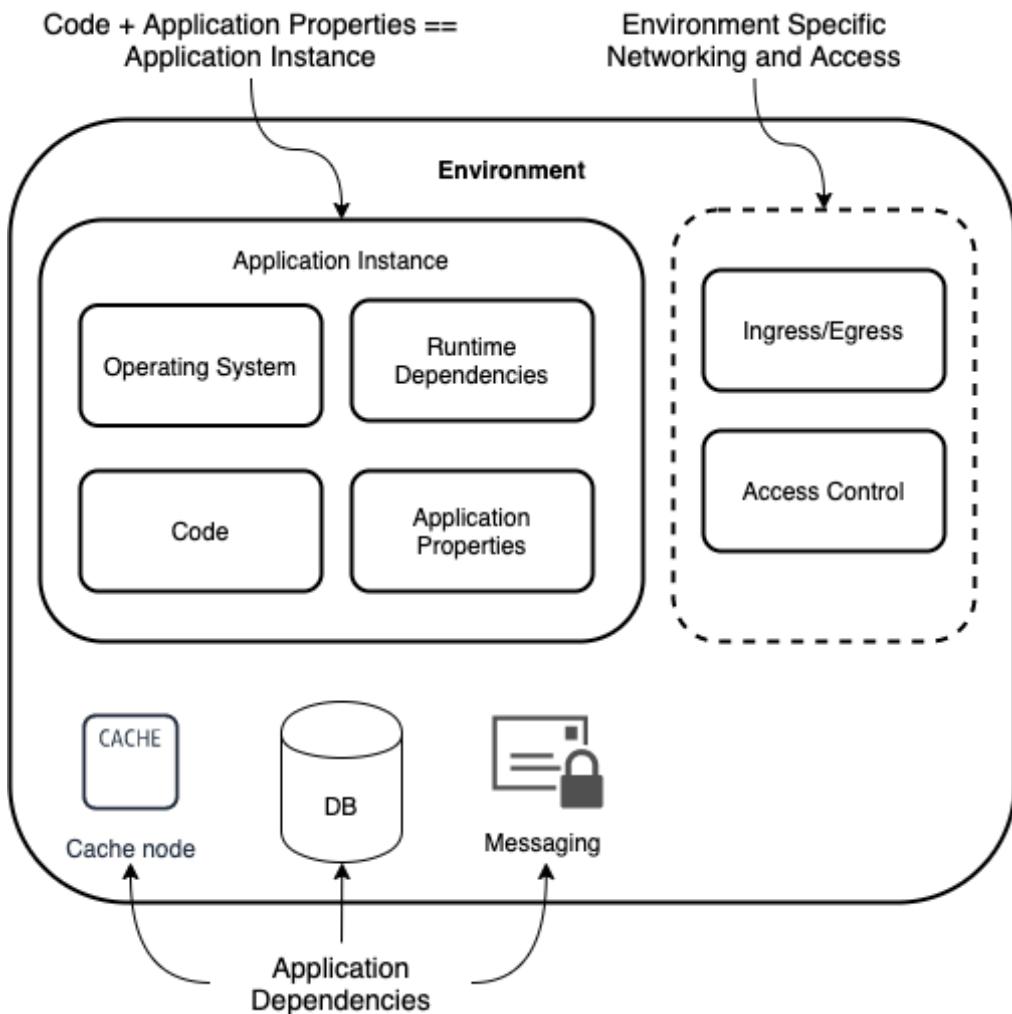


Figure 3.3 An environment will consist of application instances, Ingress/Egress for networking, and Access control to protect its resources. An environment will also include application dependencies such as Cache, DB, or Messaging.

PICKING THE “RIGHT” GRANULARITY

Ultimately, the goal is for all new code to be deployed to production so that customers and end-users can begin using it as soon as it passes quality testing. Delays in deploying code to production result in the postponement of realizing the new code’s business value produced by the development team. Picking the right environment granularity is critical for code to be deployed without delay. Factors to be considered are:

1. Release Independence: If the code needs to be bundled together with code from other teams to be deployed, one team's deployment cycle is subject to the readiness of the code produced by the other teams. The right granularity should enable your code to be deployed without dependencies on other teams/code.
2. Test Boundary: Like release independence, testing of the new code should be independent of other code releases. If new code testing depends on other teams/code, the release cycle will be subject to the readiness of the others.
3. Access Control: In addition to separate access control for pre-prod and prod, each environment can limit access control to only the team actively working on the codebase.
4. Isolation: Each environment is a logical work unit and should be isolated from other environments to avoid the "noisy neighbor" problem and limiting access from different environments for security reasons.

3.1.2 Namespace Management

Namespaces are a natural construct in Kubernetes to support environments. Namespaces allow dividing cluster resources among multiple teams or projects. Namespaces provide a scope for unique resource naming, resource quotas, Role-Based Access Control (RBAC), hardware isolation, and network configuration.

Kubernetes Namespace \approx Environment

In each namespace, the application instance (aka Pod) is one or more Docker containers injected with environment-specific application properties during deployment. These application properties define how the environment should run (e.g., feature flags) and what external dependencies should be used (e.g., database connection strings).

In addition to the application Pods, the namespace may also contain other Pods that provide additional functionality required by the environment.

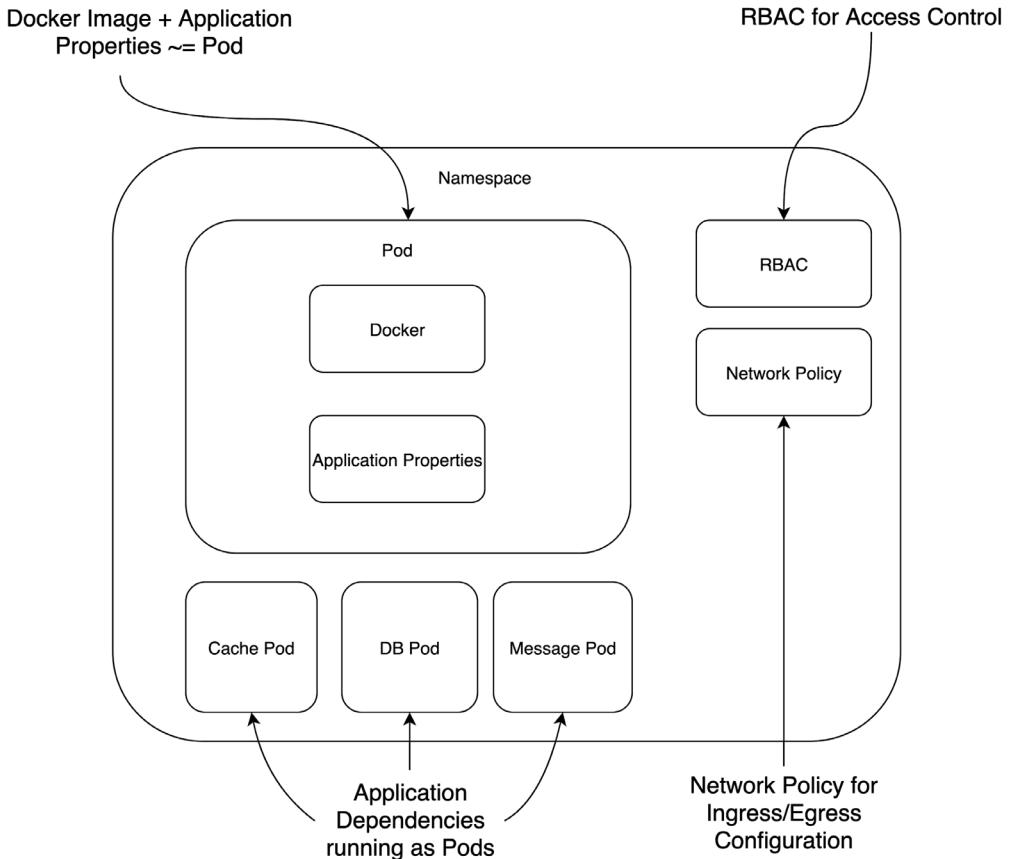


Figure 3.4 A Namespace is the equivalent of an environment in Kubernetes. Namespaces may consist of Pods (application instance), NetworkPolicies (ingress/Egress), and RBAC (Access control), as well as running application dependencies running in separate pods.

Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise. In Kubernetes, “a role contains rules that represent a set of permissions. Permissions are purely additive (there are no “deny” rules). A role can be defined within a namespace with a Role, or cluster-wide with a ClusterRole.”

Namespaces can also have dedicated hardware and networking policies to optimize their configuration based on application requirements. For example, a CPU intensive application can deploy in a namespace with dedicated multi-core hardware. Another service requiring heavy disk I/O can be deployed in a separate namespace with high-speed SSD. Each namespace can also define its networking policy (ingress/egress) to limit cross namespace traffic or accessing other namespaces within the cluster using unqualified DNS names.

DEPLOY AN APP IN TWO DIFFERENT ENVIRONMENTS

In this section, you will learn how to deploy the same app in two different environments (a test environment called “guestbook-qa” and a pre-prod end-to-end environment called “guestbook-e2e”) with different configurations using namespaces. The application we will use for this exercise is the Guestbook Kubernetes example application.¹⁵

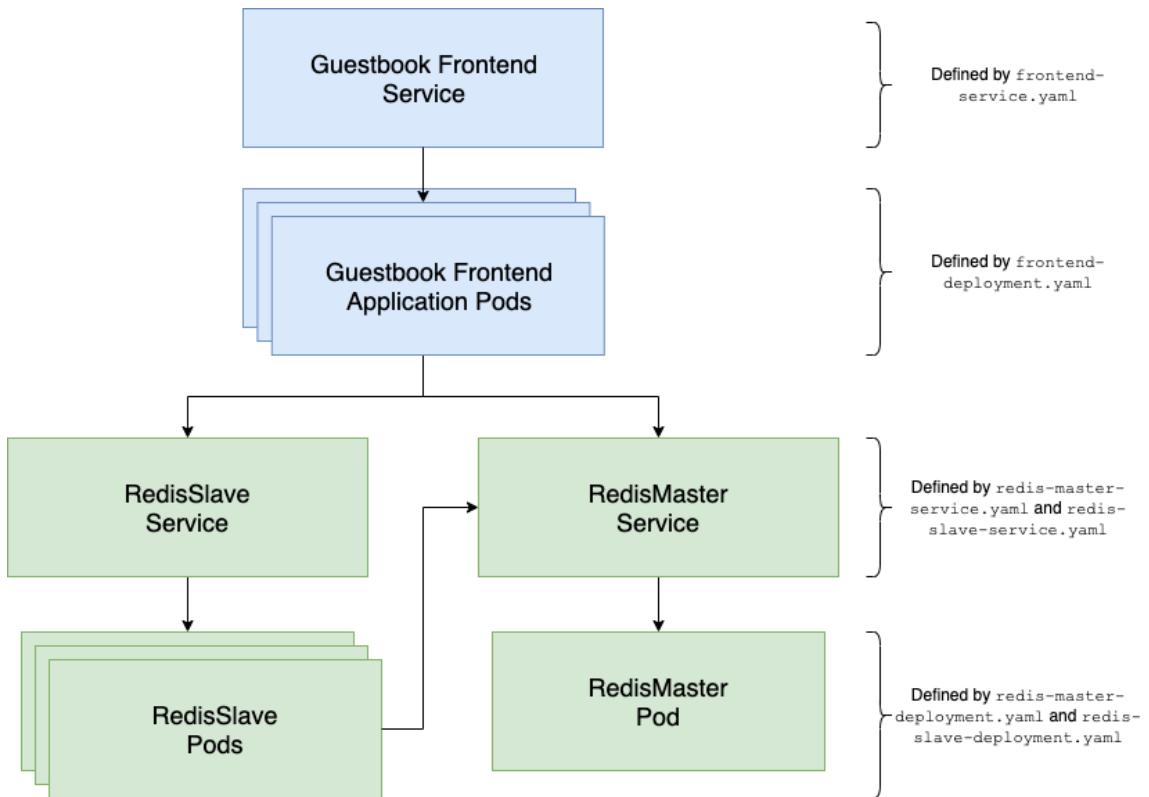


Figure 3.5 Guestbook front end architecture will have a service to expose the Guest book web front end to live traffic. The back end architecture consists of Redis Master and Redis Slave for the data.

Exercise Overview

1. Create environment namespaces (guestbook-qa and guestbook-e2e)
2. Deploy the guestbook application to the guestbook-qa environment
3. Test the guestbook-qa environment
4. “Promote” the guestbook application to the guestbook-e2e environment
5. Test the guestbook-e2e environment

¹⁵ <https://kubernetes.io/docs/tutorials/stateless-application/guestbook/>

VERIFY KUBERNETES CLUSTER CONNECTION Before you begin, verify you have correctly configured your KUBECONFIG environment variable to point to the desired Kubernetes cluster. Please refer to the “Run Kubernetes Locally” exercise in Appendix A for more information.

First, create the guestbook-qa and guestbook-e2e namespaces for each of your guestbook environments.

```
$ kubectl create namespace guestbook-qa
namespace/guestbook-qa created
$ kubectl create namespace guestbook-e2e
namespace/guestbook-e2e created
$ kubectl get namespaces
NAME           STATUS  AGE
default        Active  2m27s
guestbook-e2e  Active  9s
guestbook-qa   Active  19s
kube-node-lease Active  2m30s
kube-public    Active  2m30s
kube-system    Active  2m30s
```

Now you can deploy the guestbook application to the guestbook-qa environment using the following commands:

```
$ export K8S_GUESTBOOK_URL=https://k8s.io/examples/application/guestbook
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/redis-master-deployment.yaml
deployment.apps/redis-master created
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/redis-master-service.yaml
service/redis-master created
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/redis-slave-deployment.yaml
deployment.apps/redis-slave created
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/redis-slave-service.yaml
service/redis-slave created
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/frontend-deployment.yaml
deployment.apps/frontend created
$ kubectl apply -n guestbook-qa -f ${K8S_GUESTBOOK_URL}/frontend-service.yaml
service/frontend created
```

Before we proceed, let’s test that the `guestbook-qa` environment is working as expected. Use the following minikube command to find the URL to the `guestbook-qa` service and then open the URL in your web browser.

```
$ minikube -n guestbook-qa service frontend --url
http://192.168.99.100:31671
$ open http://192.168.99.100:31671
```

In the “Messages” text edit of the guestbook application, type something like “This is the guestbook-qa environment.” and press the “Submit” button. Your screen should look something like the following figure.

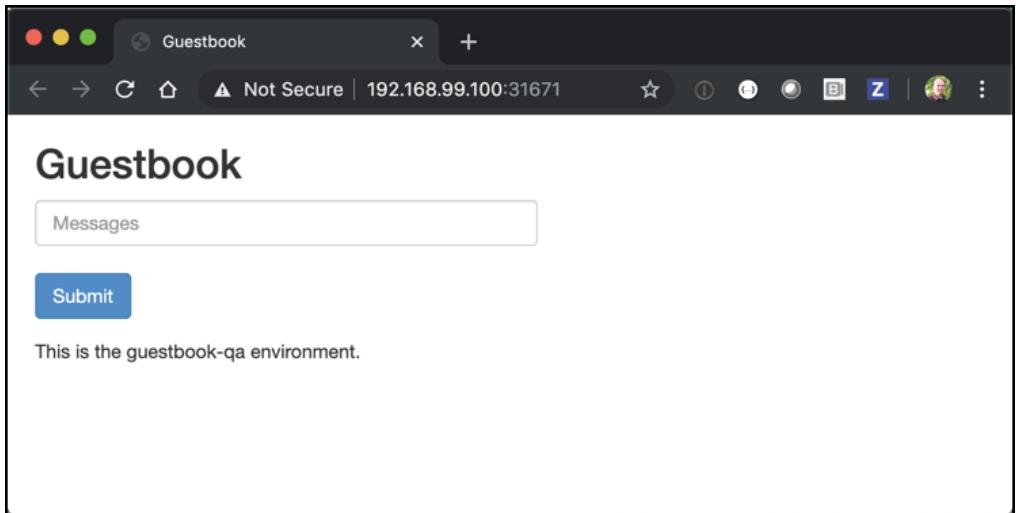


Figure 3.6 Once you have your guestbook app deployed to QA, you can open the browser and submit a test message to verify your deployment.

Now that we have the Guestbook application running in the guestbook-qa environment and tested that it is working correctly, let's "promote" the guestbook-qa to the guestbook-e2e environment. In this case we are going to use exactly the same YAML as was used in the guestbook-qa environment. This is very similar to how your automated CD pipeline would work.

```
$ export K8S_GUESTBOOK_URL=https://k8s.io/examples/application/guestbook
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/redis-master-deployment.yaml
deployment.apps/redis-master created
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/redis-master-service.yaml
service/redis-master created
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/redis-slave-deployment.yaml
deployment.apps/redis-slave created
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/redis-slave-service.yaml
service/redis-slave created
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/frontend-deployment.yaml
deployment.apps/frontend created
$ kubectl apply -n guestbook-e2e -f ${K8S_GUESTBOOK_URL}/frontend-service.yaml
service/frontend created
```

Great! The Guestbook app has now been deployed to the guestbook-e2e environment. Now, let's test the guestbook-e2e environment is working correctly.

```
$ minikube -n guestbook-e2e service frontend --url
http://192.168.99.100:31090
$ open http://192.168.99.100:31090
```

Similar to what you did in the guestbook-qa environment, type something like "This is the guestbook-e2e environment, NOT the guestbook-qa environment!" in the "Messages" text

edit and press the “Submit” button. Your screen should look something like the following figure.

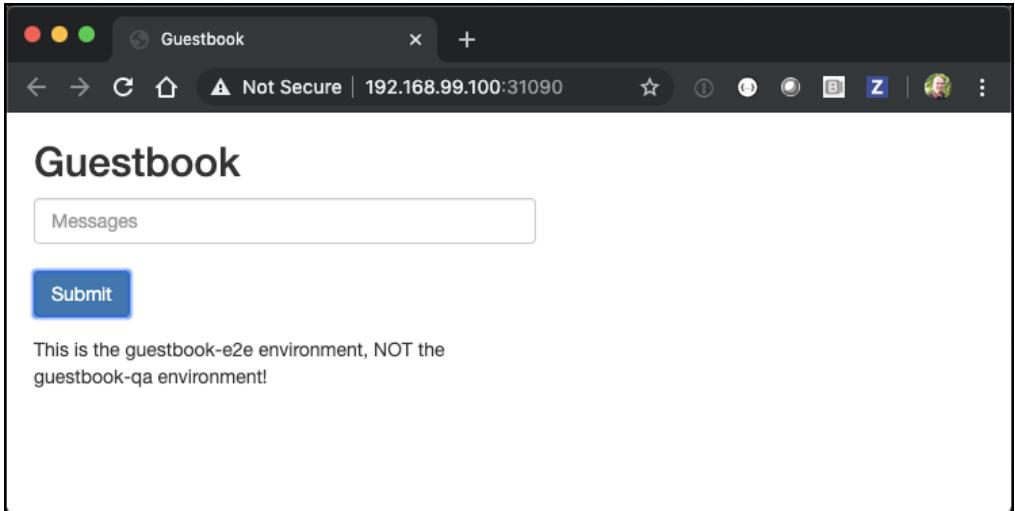


Figure 3.7 Once you have your guestbook app deployed to QA, you can open the browser and submit a test message to verify your deployment.

The important thing here is to realize that you have identical applications running in two different environments defined by Kubernetes namespaces. Notice that each application is maintaining an independent copy of its data. If you enter a message in the QA Guestbook, it doesn’t show up in the E2E Guestbook. These are two different environments.

Exercise 3.1

Now that you have created two pre-prod environments, guestbook-qa and guestbook-e2e, create two additional production environments, guestbook-stage, and guestbook-prod, in a new “production” cluster. HINT: You can create a new minikube cluster with the command `minikube start -p production` and switch between them using `kubectl config use-context <name>`.

Case Study: Intuit environment management

At Intuit, we organize our namespaces per service and per environment in each AWS region with the separation of pre-prod and prod clusters. A typical service will have six namespaces: QA, E2E, Stage/Prod West, and Stage/Prod East. QA and E2E namespaces will be in the pre-prod cluster with open access to the corresponding team. Stage/Prod West and Stage/Prod East will be in the production cluster with restricted access.¹⁸

¹⁸ <https://www.cncf.io/case-study/intuit/>

3.1.3 Network isolation

A critical aspect of defining environments for deploying your application is to ensure only the intended clients can access the specific environment. By default, all namespaces can connect to services running in all other namespaces. But in the case of two different environments, for example, QA and Prod”, you would not want cross-talk between those environments. Luckily it is possible to apply a namespace network policy that restricts network communication between namespaces. Let’s look at how we can deploy an application to two different namespaces and control access using network policies.

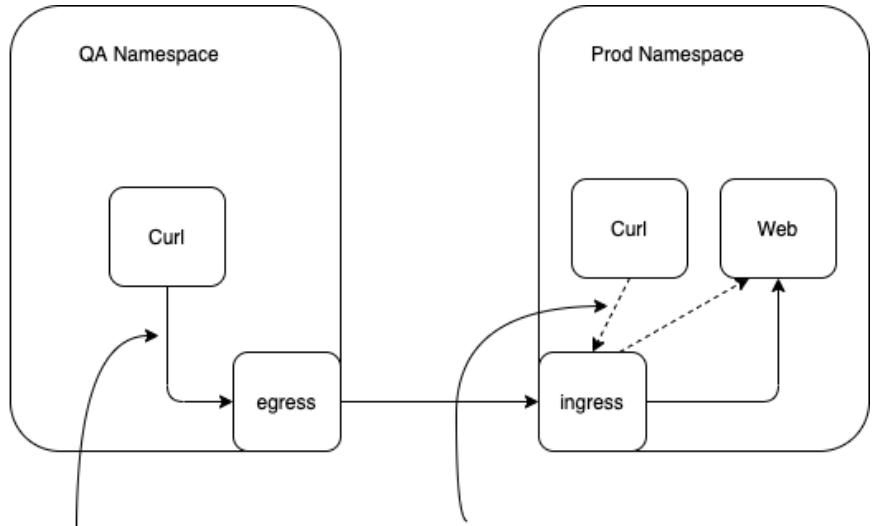
We will go over the steps to deploy services in two different namespaces. You will also modify the network policies and observe the effects.

Overview

1. Create environment namespaces (qa and prod)
2. Deploy curl to the qa and prod namespaces
3. Deploy nginx to the prod namespace
4. Curl nginx from both the qa and prod namespaces (both work)
5. Block incoming traffic to the prod namespace from qa namespace.
6. Curl nginx from the qa namespace (blocked)

EGRESS Egress traffic is network traffic that begins inside a network and proceeds through its routers to a destination somewhere outside the network.

INGRESS Ingress traffic is composed of all the data communications and network traffic originating from external networks.



Solid line illustrates that request crossing QA and Prod will be intercepted by QA egress and Prod ingress.

Dotted line illustrates that request within the Prod will only be intercepted by Prod ingress.

Figure 3.8 For the Curl “pod” in QA to reach the Web “pod” in Prod, the Curl “pod” will need to go through the QA egress to reach the Prod ingress. Then the Prod ingress will route the traffic to the Web “pod” in Prod.

VERIFY KUBERNETES CLUSTER CONNECTION Before you begin, verify you have correctly configured your KUBECONFIG environment variable to point to the desired Kubernetes cluster. Please refer to the “Run Kubernetes Locally” exercise in Appendix A for more information.

First, create the namespaces for each of your environments.

```
$ kubectl create namespace qa
namespace/qa created
$ kubectl create namespace prod
namespace/prod created
$ kubectl get namespaces
NAME      STATUS   AGE
qa        Active   2m27s
prod      Active   9s
```

Now we will create a pod in both namespaces from where we can run the Linux command curl.

```
$ kubectl -n qa apply -f curlpod.yaml
$ kubectl -n prod apply -f curlpod.yaml
```

Listing 3.1 curlpod.yaml.

```

apiVersion: v1
kind: Pod
metadata:
  name: curl-pod
spec:
  containers:
    - name: curlpod
      image: radial/busyboxplus:curl
      command:
        - sh
        - -c
        - while true; do sleep 1; done

```

In the prod namespace, we will run an nginx server that will receive the curl HTTP request.

```
$ kubectl -n prod apply -f web.yaml
```

Listing 3.2 web.yaml.

```

apiVersion: v1
kind: Pod
metadata:
  name: web
spec:
  containers:
    - image: nginx
      imagePullPolicy: Always
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP

```

By default, pods running in a namespace can send network traffic to other pods running in different namespaces. Let's prove this by executing a curl command from the pod in the qa namespace to the nginx pod in the prod namespace.

```
$ kubectl describe pod web -n prod | grep IP #A
$ kubectl -n qa exec curl-pod -- curl -I http://<web pod ip> #B
$ kubectl -n prod exec curl-pod -- curl -I http://<web pod ip> #C
```

#A Get the web pod IP address

#B Get back HTTP 200

#C Get back HTTP 200

Typically you never want your qa and prod environments to have dependencies between each other. It may be that if both instances of the application were properly configured, there would not be dependencies between qa and prod, but what if there was a bug in the configuration of qa where it was accidentally sending traffic to prod? You could potentially be corrupting production data. Or even within production, what if one environment was hosting your marketing site and another environment was hosting an HR application with sensitive data? In these cases, it may be appropriate to block network traffic between namespaces or only allow network traffic between particular namespaces. This can be accomplished by adding a `NetworkPolicy` to a namespace.

Let's add a `NetworkPolicy` to our pods in each namespace.

CONTAINER NETWORK INTERFACE Network Policy is only supported if Container Network Interface (CNI)⁴⁷ is configured (neither minikube nor Docker desktop). Please refer to the “Create a GKE cluster” or “Create an EKS Cluster in AWS” in Appendix A for more information to test the configuration of network policies.

```
$ kubectl apply -f block-other-namespace.yaml
```

Listing 3.3 Network Policy ([block-other-namespace.yaml](#)).

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  namespace: prod                         #A
  name: block-other-namespace
spec:
  podSelector: {}                          #B
  ingress:
    - from:
        - podSelector: {}                  #C
```

#A Apply to namespace `prod`

#B Select all pods in namespace `prod`

#C Specify ingress only to allow requests coming from `prod` namespace only. Requests from other namespaces will be blocked.

This `NetworkPolicy` is applied to the `prod` namespace and only allows ingress (incoming network traffic) from the `prod` namespace.

Correctly utilizing `NetworkPolicy` constraints are a critical aspect of defining environment boundaries.

With the `NetworkPolicy` applied, we can rerun our `curl` commands to verify that each namespace is now isolated from the other.

```
$ kubectl -n qa exec curl-pod -- curl -I http://<web pod ip>      #A
$ kubectl -n prod exec curl-pod -- curl -I http://<web pod ip>      #B
```

#A Curl from namespace “qa” is blocked!

#B Get back Http 200

3.1.4 Pre-prod and Prod Clusters

Now that you know how to create multiple environments using Namespaces, it might seem like a trivial thing to use one cluster and create all the environments you need on that single cluster. For example, you may need QA, E2E, Stage, and Prod environments for your application. However, depending on your specific use-case, this may not be the best approach. Our recommendation is to have two clusters to host your environments; one “pre-prod cluster” for pre-production environments and one “prod cluster” for production environments.

⁴⁷ <https://www.cncf.io/blog/2017/05/23/cncf-hosts-container-networking-interface-cni/>

The primary reason for having two separate clusters to host your environments is to protect your production environment from accidental outages or other impacts related to work being done with the pre-production environments.

CLUSTER ISOLATION IN AWS In AWS, a separate VPC can be created for pre-prod and prod as a logical boundary to isolate the traffic and data. For even stronger isolation and more control on production credentials and access, the separate “production” VPC should be hosted in a different “production” AWS account.

Someone might ask why we should have so many environments and separation of pre-prod and prod clusters. The simple answer is that a pre-prod cluster is needed to test the code before release into the production cluster. At Intuit, we use our QA environment for integration testing and E2E environment as a stable environment for other services to test pre-release features. If you are doing multi-branch concurrent development, you can also configure additional pre-prod test environments for each of the branches.

A key advantage of configuration management with Kubernetes is that since it uses Docker Containers, which are immutable portable images, the only difference with the deployments between environments is the namespace configuration, environment-specific properties, and application dependencies such as caching or database. Pre-prod testing can verify your service code’s correctness, while the stage environment in the production cluster can be used to verify your application dependencies’ correctness.

Pre-prod and prod clusters should follow the same security best practices and operational rigor. Security issues can be detected early in the development cycle, and developer productivity is not interrupted if pre-prod clusters are operated with the same standards as production.

3.2 Git Strategies

Using a separate Git repository to hold your Kubernetes manifests (aka config), keeping the config separate from your application source code, is highly recommended for the following reasons:

1. It provides a clean separation of application code vs. application config. There will be times when you wish to modify just the manifests without triggering an entire CI build. For example, you likely do not want to trigger a build if you simply want to bump the number of replicas in a Deployment spec.

NOTE Application Config vs. Secrets In GitOps, application configuration generally excludes secrets since using git to store secrets is a bad practice. There are several approaches for handling sensitive information (e.g., passwords, certificates, etc.), discussed in detail in chapter 7, Secret Management.

2. Cleaner audit log. For auditing purposes, a repo which only holds configuration will have a much cleaner Git history of what changes were made, without the noise coming from check-ins due to regular development activity.

3. Your application may comprise services built from multiple Git repositories but is deployed as a single unit. Frequently, microservices applications are composed of services with different versioning schemes and release cycles (e.g., ELK, Kafka + Zookeeper). It may not make sense to store the manifests in one of the source code repositories of a single component.
4. Separation of access. The developers who are developing the application may not necessarily be the same people who can/should push to production environments, either intentionally or unintentionally. Having separate repositories allows commit access can be given to the source code repo and not the application config repo, which can be reserved for a more select group of team members.
5. If you are automating your CI pipeline, pushing manifest changes to the same Git repository can trigger an infinite loop of build jobs and Git commit triggers. Having a separate repo to push config changes to prevents this from happening.

For your code repositories, you can use whatever branching strategy you like (e.g., GitFlow) since it is only used for your CI. For your config repositories (which will be used for your CD), you need to consider the following strategies based on your organization size and tooling.

3.2.1 Single branch (multiple directories)

With the single branch strategy, the main branch will always contain the exact config used in each environment. There will be a default config for all environments with an environment-specific overlay defined in separate environment-specific directories. The single branch strategy can be easily supported with tools such as Kustomize (we will discuss in detail in section 3.3).

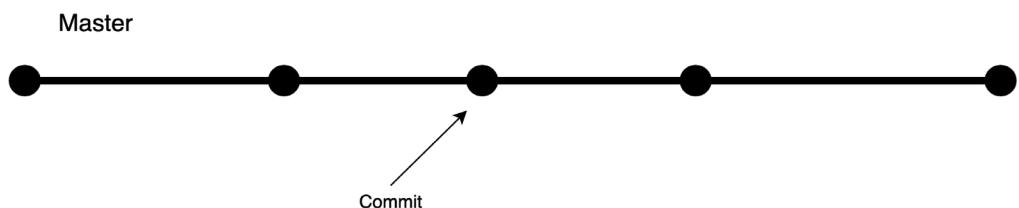


Figure 3.9 Single branch strategy will have one master branch and a subdirectory for each environment. Each subdirectory will contain the environment-specific overlay.

In our CI/CD example, we will have environment-specific override directories for `qa`, `e2e`, `stage`, and `prod`. Each directory will contain environment-specific settings such as replica count, CPU, and memory request/limit.

Branch: master ▾ gitops---k8s-deployment / environments /	
 svcdoadmin	Initial commit
..	
└─ e2e-usw2	Initial commit
└─ prd-usw2	Initial commit
└─ qal-usw2	Initial commit
└─ stg-usw2	Initial commit

Figure 3.10 This is an example with qal, e2e, stage, and production sub-directories. Each sub-directory will contain overlays such as replica count, CPU, and memory request/limit.

3.2.2 Multiple branches

With the multiple branches strategy, each branch is equivalent to an environment. The advantage here is that each branch will have the exact manifest for the environment without using any tool such as Kustomize. Each branch will also have separate commit history for audit trail and rollback if needed. The disadvantage is that there will be no sharing of the common config among environments since tooling such as Kustomize do not work with Git Branches.

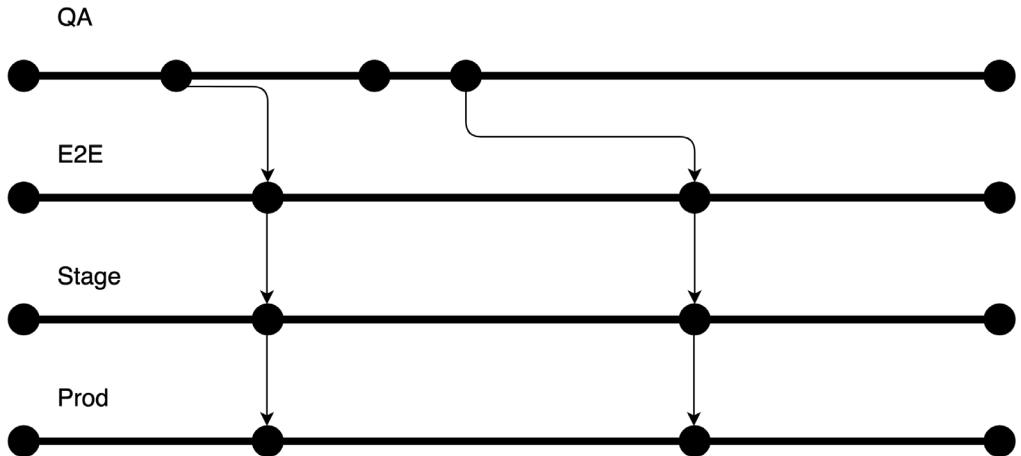


Figure 3.11 In the multiple branches strategy, each branch is equivalent to an environment. Each branch will contain the exact manifest instead of an overlay.

It may be possible to merge common infrastructure changes between multiple branches. For example, suppose a new resource needs to be added to all environments. In that case, that resource could be added first to the QA branch, tested, and then merged (e.g., “cherry-picked”) into each successive branch after the appropriate testing was completed.

3.2.3 Multiple repos vs. Monorepo

If you are in a startup environment with a single scrum team, you may not want (or need) the complexity of multiple repositories. All your code could be in one code repository with all your deployment configuration in one deployment repository.

However, if you are in an enterprise environment with dozens (or hundreds) of developers, you will likely want to have multiple repos so that teams can be decoupled from each other and each run at their own speed. For example, different teams within the organization will have a different cadence and release process for their code. If mono config repo is used, some features may be completed for weeks but need to wait for the “scheduled release.” This could mean delaying getting features into the hands of end-users and discovering potential code issues. Rollback is also problematic since one code defect will require rolling back all changes from every team.

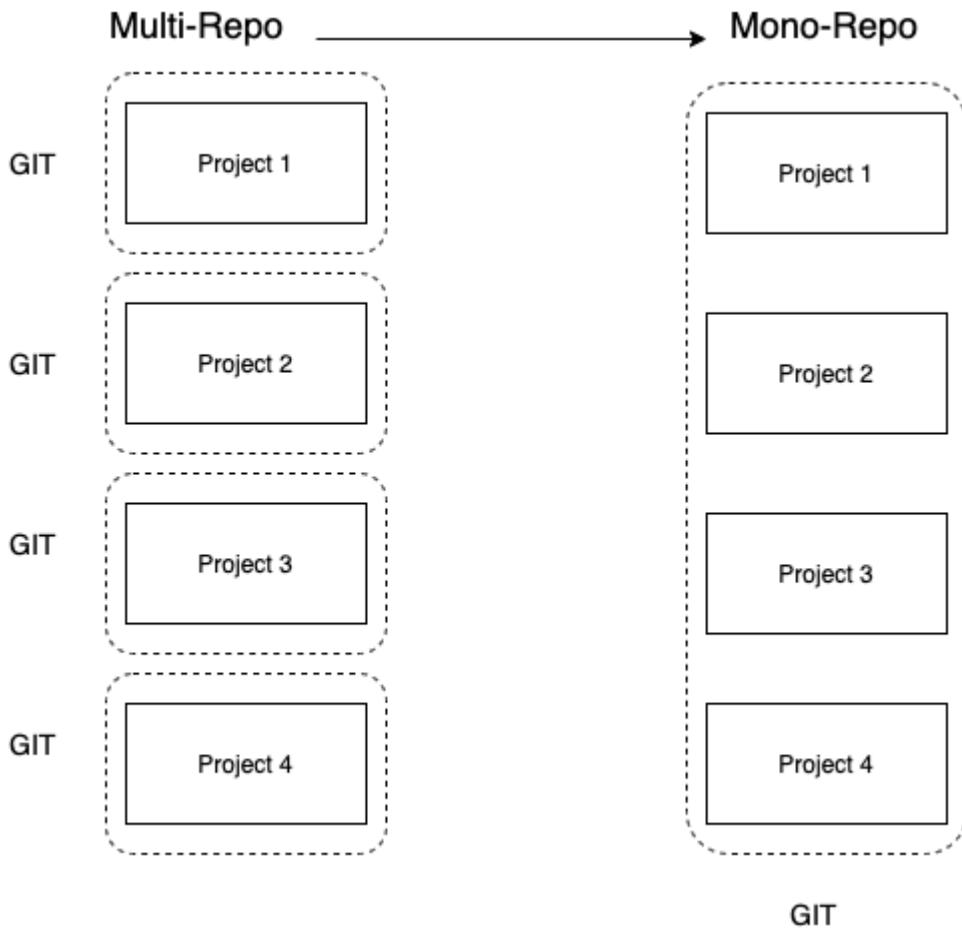


Figure 3.12 Mono repo is a single Git repo with multiple projects. In Multi-repo, each project will have a dedicated Git repo.

Another consideration to use multiple repos is to organize applications based on “capabilities.” If the repos are focused on discrete deployable capabilities, it will be easier to move the responsibility of those capabilities among teams (e.g., after a reorg).

3.3 Configuration Management

As we saw in the tutorial and exercises in section 3.1, environment configuration management can be as simple as having a directory for each environment that contains the YAML manifests of all resources that should be deployed. All the values in these YAML manifests can be hard-coded to the particular values desired for that environment. To deploy, you just run `kubectl apply -f <directory>`.

However, the reality is managing multiple configurations in that way becomes unwieldy and error-prone quickly. What if you need to add a new resource? You need to make sure you add that resource to the YAML in each of your environments. What if that resource needs specific properties (like replicas) to have different values for different environments? You need to carefully make all the correct customizations in all the right files.

Thankfully several tools have been developed that address this need for configuration management. We will review each of the more popular configuration management tools later in this section. But first, let's discuss what the factors are that you should consider when choosing which particular tool to use.

Good Kubernetes configuration tools have the following properties:

Declarative: The config is unambiguous, deterministic, and not system dependent.

Readable: The config is written in a way that is easy to understand.

Flexible: The tool helps facilitate and does not get in the way of accomplishing what you are trying to do.

Maintainable: The tool should promote re-use and composability.

There are several reasons why Kubernetes config management is so challenging: what sounds like a simple act of deploying an application can have wildly different, even opposing requirements, and it's difficult for a single tool to accommodate all such requirements. Imagine the following use cases:

A cluster operator who deploys 3rd-party, off-the-shelf applications, such as WordPress, to their cluster with little to no customization of those apps: The most important criteria for this use-case is to easily receive updates from an upstream source and upgrade their application as quickly and seamlessly as possible (e.g., new versions, security patches, etc.).

A Software as a Service (SaaS) application developer who deploys their bespoke application to one or more environments (e.g., dev, staging, prod-west, prod-east): These environments may be spread across different accounts, clusters, and namespaces with subtle differences between them, so configuration re-use is paramount. For this use-case, it is important to go from a Git commit in their codebase to deploy to each of their environments in a fully automated way and manage their environments in a straightforward and maintainable way. These developers have zero interest in semantic versioning of their releases since they might be deploying multiple times a day. The notion of major, minor, and patch versions ultimately has no meaning for their application.

As you can see, these are entirely different use-cases, and more often than not, a tool that excels at one doesn't handle the other very well.

3.3.1 Helm

Love it or hate it, Helm, being the first config tool on the scene, is an integral part of the Kubernetes ecosystem, and the chances are that you've installed something at one point or another by running `helm install`.

The important thing to note about Helm is that it is a self-described package manager for Kubernetes and doesn't claim to be a configuration management tool. However, since many people use Helm templating for precisely this purpose, it belongs in this discussion. These users invariably end up maintaining several `values.yaml`, one for each environment (e.g., `values-base.yaml`, `values-prod.yaml`, `values-dev.yaml`), then parameterize their chart in

such a way that environment-specific values can be used in the chart. This method more or less works, but it makes the templates unwieldy since golang templating is flat and needs to support every conceivable parameter for each environment, which ultimately litters the entire template with `{}-if / else{}` switches.

The Good:

There's a chart for that. Undoubtedly, Helm's biggest strength is its excellent chart repository. Just recently, we needed to run a highly available Redis, without a persistent volume, to be used as a throwaway cache. There is something to be said about just being able to throw the `redis-ha` chart into your namespace, set `persistentVolume.enabled: false`, point your service at it, and someone else has already done the hard work of figuring out how to run Redis reliably on a Kubernetes cluster.

The Bad:

Golang templating. "Look at that beautiful and elegant Helm template!" said no one ever. It is well known that Helm templates suffer from a readability problem. We don't doubt that this will be addressed with Helm 3's support for Lua, but until then, well, I hope you like curly braces.

Complicated SaaS CD pipelines. For SaaS CI/CD pipelines, assuming you are using Helm the way it is intended (i.e., by running `helm install/upgrade`), an automated deploy in your pipeline might go several ways. In the best case, deploying from your pipeline will be as simple as:

```
$ docker push mycompany/guestbook:v2
$ helm upgrade guestbook --set guestbook.image.tag=v2
```

But in the worst case, where existing chart parameters cannot support your desired manifest changes, you go through a whole song and dance of bundling a new Helm chart, bumping its semantic version, publishing it to a chart repository, and redeploying with a `helm upgrade`. In the Linux world, this is analogous to building a new RPM, publishing the RPM to a yum repository, then running yum install, all so you can get your shiny new CLI into `/usr/bin`. While this model works great for packaging and distribution, it's an unnecessarily complicated and roundabout way to deploy your applications in the case of bespoke SaaS applications. For this reason, many people choose to run `helm template` and pipe the output to `kubectl apply`, but at that point, you are better off using some other tool that is specifically designed for this purpose.

Non-declarative by default. If you ever added `--set param=value` to any one of your Helm deploys, I'm sorry to tell you that your deployment process is not declarative. These values are only recorded in the Helm ConfigMap netherworld (and maybe your bash history), so hopefully you wrote those down somewhere. This is far from ideal if you ever need to recreate your cluster from scratch. A slightly better way would be to record all parameters in a new custom `values.yaml` that you can store in Git and deploy using `-f my-values.yaml`. However, this is annoying when you're deploying an OTS chart from Helm stable, and you don't have an obvious place to store that `values.yaml` side-by-side to the relevant chart. The best solution that I've come up with is composing a new dummy chart with the upstream

chart as a dependency. Still, I have yet to find a canonical way of updating a parameter in a `values.yaml` in a pipeline using a one-liner, short of running `sed`.

CONFIGURE MANIFESTS FOR PRE-PROD VS. PROD USING HELM

In this exercise, we will take the guestbook app we deployed earlier in the chapter and use Helm to manage its configuration for different environments.

Helm uses the following directory structure to structure its charts.

Listing 3.4 Helm Chart Directory Structure.

```
.
├── Chart.yaml          #A
└── templates
    └── guestbook.yaml
├── values-prod.yaml   #C
└── values-qa.yaml      #C
```

#A The `Chart.yaml` is Helm's descriptor of the chart.

#B A directory of templates that, when combined with `values`, will generate valid Kubernetes manifest files.

#C Various configuration values for the chart, which can be used for environment-specific configuration

A Helm template file uses a text templating language to generate Kubernetes yaml. Helm template files look like Kubernetes YAML, but with template variables sprinkled throughout the file. As a result, even a very basic Helm template file ends up looking like this:

Listing 3.5 Sample App Helm Template.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "sample-app.fullname" . }}
  labels:
    {{- include "sample-app.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "sample-app.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      {{- with .Values.podAnnotations }}
        annotations:
          {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "sample-app.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default
            .Chart.AppVersion }}"
          {{- with .Values.environmentVars }}
            env:
              {{- toYaml . | nindent 12 }}
          {{- end }}
```

As you can see, helm templates are not very readable. But they are incredibly flexible since the final resulting yaml can be customized in any way the user desires.

Finally, when customizing a specific environment using helm charts, an environment-specific values file is created containing the values to use for that environment. For example, for the production version of this application, the values file may look like:

Listing 3.6 Sample App Helm Values.

```
# Default values for sample-app.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

image:
  repository: gitopsbook/sample-app
  tag: "v0.2"                                     #A

nameOverride: "sample-app"
fullnameOverride: "sample-app"

podAnnotations: {}

environmentVars: [                                #B
  {
    name: "DEBUG",
    value: "true"
  }
]

]                                                 #A Overrides the image tag whose default is the chart appVersion.
#B Set the DEBUG environment variable to true
```

The final qa manifest can be installed to minikube in the qa-heml namespace with the following commands.

```
$ kubectl create namespace qa-helm
$ helm template . --values values.yaml | kubectl apply -n qa-helm -f -
deployment.apps/sample-app created
$ kubectl get all -n qa-helm
NAME                           READY   STATUS    RESTARTS   AGE
pod/sample-app-7595985689-46fbj   1/1     Running   0          11s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sample-app     1/1     1           1           11s

NAME                           DESIRED   CURRENT   READY   AGE
replicaset.apps/sample-app-7595985689   1         1         1        11s
```

Exercise 3.2

In the tutorial above, we parameterized the guestbook image tag for QA and prod environments using Helm. Add additional parameterization for the number of replicas desired for each guestbook deployment. Set the number of replicas to 1 for QA and 3 for prod.

3.3.2 Kustomize

Kustomize was created around the design principles described in Brian Grant's excellent dissertation regarding Declarative Application Management¹⁸. Kustomize has seen a meteoric rise in popularity, and, in the eight months since it started, it has already been merged into kubectl. Whether or not you agree with how it was merged, it goes without saying that kustomize applications will now have a permanent mainstay in the Kubernetes ecosystem and will be the default choice that users will gravitate towards for config management. Yes, it helps to be part of kubectl!

The Good:

No parameters & templates. Kustomize apps are extremely easy to reason about, and I dare say, a pleasure to look at. It's about as close as you can get to Kubernetes YAML since the overlays you compose to perform customizations are simply subsets of Kubernetes YAML.

The Bad:

No parameters & templates. The same property that makes kustomize applications so readable can also make it very limiting. For example, I was recently trying to get the kustomize CLI to set an image tag for a custom resource instead of a Deployment but was unable to. Kustomize does have a concept of "vars," which look a lot like parameters but somehow aren't, and can only be used in Kustomize's sanctioned whitelist of field paths. I feel like this is one of those times when the solution, despite making the hard things easy, ends up making the easy things hard.

CONFIGURE MANIFESTS FOR PRE-PROD VS. PROD USING KUSTOMIZE

In this exercise, we will use a sample application that we will use later in the Tools chapters and use kustomize to deploy it.

We will organize our configuration files into the following directory structure.

Listing 3.7 Kustomize Directory Structure.

```
.
  └── base
      ├── deployment.yaml
      └── kustomization.yaml
  └── envs
      └── prod
          └── kustomization.yaml
      └── qa
          ├── debug.yaml
          └── kustomization.yaml
```

#A The base directory contains the common configuration that will be shared by the different environments

#B The envs/prod directory contains the configuration for the production environment

#C The envs/qa directory contains the configuration for the qa environment

¹⁸ <https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/declarative-application-management.md>

The manifests in the base directory contain all the resources that are common to all environments. In this simple example, we have a single Deployment resource:

Listing 3.8 Base Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - command:
          - /app/sample-app
        image: gitopsbook/sample-app:REPLACEME #A
        name: sample-app
      ports:
        - containerPort: 8080
```

#A The image defined in the base config is irrelevant. This version of the image will never be deployed since the child overlay environments in this example will override this value.

To use the “base” directory as a base to other environments, a `kustomization.yaml` must be present in the directory. The following is the simplest `kustomization.yaml` possible. It merely lists the `guestbook.yaml` as the single resource comprising the application:

Listing 3.9 Base Kustomization.

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

resources:
- deployment.yaml
```

Now that we’ve established the `kustomize` base directory, we can start customizing our environments. To customize and modify resources for a specific environment, we define an “overlay” directory that contains all the patches and customizations we want to be applied on top of the base resources. Our first overlay is the `envs/qa` directory. Inside this directory is another `kustomization.yaml` that specifies the patches that should be applied on top of the base. The following two listings provide an example of a `qa` overlay which:

1. Sets a different guestbook image to be deployed to a new tag (v0.2)
2. Adds an environment variable, `DEBUG=true`, to the guestbook container

Listing 3.10 QA Environment Kustomization.

```

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

bases:                      #A
- ../../base

patchesStrategicMerge:
- debug.yaml                #B

images:                      #C
- name: gitopsbook/sample-app
  newTag: v0.2

```

#A bases references the “base” directory that contains the shared configuration
#B debug.yaml is a reference to a kustomize patch which will modify the sample-app Deployment object and set the DEBUG environment variable
#C images override any container images defined in the base, with different tags or image repositories. This example overrides the image tag REPLACEME, with v0.2

Notice that kustomize patches look very similar to actual Kubernetes resources. This is because they are, in fact, incomplete versions of them.

Listing 3.11 QA Environment Debug Patch.

```

apiVersion: apps/v1      #A
kind: Deployment
metadata:
  name: sample-app
spec:
  template:
    spec:
      containers:
        - name: sample-app    #B
          env:
            - name: DEBUG
              value: "true"

```

#A The apiVersion group (apps), the kind (Deployment), and name (sample-app) are key pieces of information that inform Kustomize about which resource in the base this patch should be applied to.
#B The name field is used to identify which container will have the new environment variables
#C Finally, the new DEBUG environment variable we want in the “qa” environment is defined

After all that is said and done, we run kustomize build envs/qa. This produces the final, rendered manifests for the “qa” environment:

Listing 3.12 kustomize build envs/qa.

```

$ kustomize build envs/qa
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:

```

```

    app: sample-app
template:
  metadata:
    labels:
      app: sample-app
spec:
  containers:
    - command:
        - /app/sample-app
      env:
        - name: DEBUG
          value: "true"
      image: gitopsbook/sample-app:v0.2
      name: sample-app
      ports:
        - containerPort: 8080

```

#A The `DEBUG` environment variable is added

#B The image tag is set to `v0.2`

The final qa manifest can be installed to minikube in the qa namespace with the following commands.

```

$ kubectl create namespace qa
$ kustomize build envs/qa | kubectl apply -n qa -f -
# kubectl get all -n qa
NAME                 READY   STATUS    RESTARTS   AGE
pod/sample-app-7595985689-46fbj   1/1     Running   0          11s

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sample-app   1/1       1           1          11s

NAME           DESIRED   CURRENT   READY   AGE
replicaset.apps/sample-app-7595985689   1         1         1          11s

```

Exercise 3.3

In the tutorial above, we parameterized the guestbook image tag for QA and prod environments using Kustomize (Hint: create a `replica_count.yaml` patch file). Add additional parameterization for the number of replicas desired for each sample-app deployment. Set the number of replicas to 1 for QA and 3 for prod. Deploy the qa environment to the qa namespace and the prod environment to the prod namespace.

Exercise 3.4

Currently, the prod environment runs version v0.1 of the sample app, and qa runs version v0.2. Let's assume we have completed the testing in QA. Update the `customization.yaml` file to promote version v0.2 to run in prod. Update the prod environment in the prod namespace.

3.3.3 Jsonnet

Jsonnet is actually a language and not really a “tool.” Furthermore, its use is not specific to Kubernetes (although it’s been popularized by Kubernetes). The best way to think of Jsonnet is as a super-powered JSON combined with a sane way to do templating. Jsonnet combines

all the things you wish you could do with JSON (comments, text blocks, parameters, variables, conditionals, file imports), without any of the things that you hate about golang/Jinja2 templating, and adds features that you didn't even know you needed or wanted (functions, object orientation, mixins). It does all of this in a declarative and hermetic (code as data) way.

When we look at a basic Jsonnet file, it looks very similar to JSON, which makes sense because Jsonnet is a superset of JSON. All JSON is valid Jsonnet. But notice in our example, we can also have comments to the document. If you have been managing configuration in JSON for long enough, you would immediately understand how useful this is!

Listing 3.13 basic.jsonnet

```
{
    // Look! It's JSON with comments!
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
        "name": "nginx"
    },
    "spec": {
        "selector": {
            "matchLabels": {
                "app": "nginx"
            }
        },
        "replicas": 2,
        "template": {
            "metadata": {
                "labels": {
                    "app": "nginx"
                }
            },
            "spec": {
                "containers": [
                    {
                        "name": "nginx",
                        "image": "nginx:1.14.2",
                        "ports": [
                            {
                                "containerPort": 80
                            }
                        ]
                    }
                ]
            }
        }
    }
}
```

Continuing with the example, let's see how we can start leveraging simple Jsonnet features. One of the simplest ways to reduce repetition, and better organize your code/configuration, is using **variables**. In our next example, we declare a few variables at the top of the jsonnet file (name, version, and replicas) and reference those variables throughout the document. This allows us to make changes at a single, visible place, without resorting to scanning the

entire document for all other areas that need that same change, which would be error-prone, especially in large documents.

Listing 3.14 variables.jsonnet

```
local name = "nginx";
local version = "1.14.2";
local replicas = 2;
{
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
        "name": name
    },
    "spec": {
        "selector": {
            "matchLabels": {
                "app": name
            }
        },
        "replicas": replicas,
        "template": {
            "metadata": {
                "labels": {
                    "app": name
                }
            },
            "spec": {
                "containers": [
                    {
                        "name": name,
                        "image": "nginx:" + version,
                        "ports": [
                            {
                                "containerPort": 80
                            }
                        ]
                    }
                ]
            }
        }
    }
}
```

Finally, with our advanced example, we start leveraging a few of the unique and powerful features of Jsonnet: functions, arguments, references, and conditionals. The next example starts to demonstrate the power of Jsonnet:

Listing 3.15 advanced.jsonnet.

```
function(prod=false) { # A, B
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
        "name": "nginx"
    },
    "spec": {
```

```

"selector": {
    "matchLabels": {
        "app": $.metadata.name      # C
    }
},
"replicas": if prod then 10 else 1, # D
"template": {
    "metadata": {
        "labels": {
            "app": $.metadata.name
        }
    },
    "spec": {
        "containers": [
            {
                "name": $.metadata.name,
                "image": "nginx:1.14.2",
                "ports": [
                    {
                        "containerPort": 80
                    }
                ]
            }
        ]
    }
}
}

```

#A Unlike previous examples, the configuration is defined as a Jsonnet **function** instead of a normal Jsonnet object.

This allows the configuration to declare inputs and accept arguments from the command line.

#B prod is a boolean argument to the function, with a default value of false

#C We can self-reference other parts of the document without using variables

#D The number of replicas can switch values change based on a condition

Exercise 3.5

In listing 3.15, try running the following two commands and compare the output:

```
$ jsonnet advanced.jsonnet
$ jsonnet --tla-code prod=true advanced.jsonnet
```

There are many more language features in Jsonnet, and we haven't even scratched the surface of its capabilities. Jsonnet is not widely adopted in the Kubernetes community, which is unfortunate because, of all the tools described here, Jsonnet is hands down the most powerful configuration tool available and is why several offshoot tools are built on top of it. Explaining what's possible with Jsonnet is a chapter in and of itself, which is why we encourage you to read how Databricks uses Jsonnet with Kubernetes, and Jsonnet's excellent learning tutorial.¹⁹

The Good:

¹⁹ <https://databricks.com/blog/2017/06/26/declarative-infrastructure-jsonnet-template-language.html>

Extremely powerful. It's rare to hit a situation that couldn't be expressed in some concise and elegant snippet of Jsonnet. With Jsonnet, you are continually finding new ways to maximize re-use and avoid repeating yourself.

The Bad:

It's not YAML. This might just be an issue with unfamiliarity, but most people will experience some level of cognitive load when they're staring at a non-trivial Jsonnet file. In the same way you would need to run a helm template to verify your Helm chart is producing what you expect, you will similarly need to run `jsonnet --yaml-stream guestbook.jsonnet` to verify your Jsonnet is correct. The good news is that, unlike with golang templating that can produce syntactically incorrect YAML due to some misplaced whitespace, these types of errors are caught with Jsonnet during the build, and the resulting output is guaranteed to be valid JSON/YAML.

ksonnet Not to be confused with Jsonnet, ksonnet²⁰ is a defunct tool for creating application manifests that can be deployed to a Kubernetes cluster. However, ksonnet is no longer maintained, and other tools should be considered instead.

3.3.4 Configuration Management Summary

As with everything, there are tradeoffs to using each tool. Below is a summary of how these specific tools compare in terms of the four qualities we value in configuration management:

Table 3.1 Features Comparison

	Helm	Kustomize	Jsonnet
Declarative	Fair	Excellent	Excellent
Readability	Poor	Excellent	Fair
Flexibility	Excellent	Poor	Excellent
Maintainability	Fair	Excellent	Excellent

Note that the tools discussed in this chapter are just the ones that happen to be the most popular in the Kubernetes community at the time of writing. This is a continually evolving space, and there are many other configuration management tools to consider.

²⁰ <https://ksonnet.io/>

3.4 Durable vs. Ephemeral Environments

Durable environments are environments that will always be available. For example, the production environment always needs to be available so that services do not get interrupted. In a durable environment, resources (memory, CPU, storage) will be committed permanently to achieve always-on availability. Often E2E is a durable environment for internal integration, and prod is a durable environment for production traffic.

Ephemeral environments are temporary environments that are not relied on by other services. Ephemeral environments also do not require resources to be permanently committed. For example, Stage is used for testing production readiness for the new code and does not need to be around after testing is complete. Another use case is for “previewing the Pull Request” for correctness to guarantee only “good code” is merged into master. In this case, a temporary environment will be created with the pull request changes so it can be tested. Once all testing is complete, the PR environment will be deleted, and the PR changes will only be allowed to be merged back to master if all tests pass.

ROLLBACK OF DURABLE ENVIRONMENTS

Given durable environments will be used by others, defects in the durable environment could interrupt others and might require rollback to restore the correct functionality. With GitOps and Kubernetes, Rollback is simply to re-apply the previous config thru Git. Kubernetes will detect the changes in the manifest and restore the environment to the previous state.

Kubernetes makes rollback in environments consistent and straightforward, but what about other resources like databases? Since user data is stored in the database, we cannot simply roll back the database to a previous snapshot, resulting in the loss of user data. As with the Rolling updates deployment in Kubernetes, new and old versions of the code need to be compatible to enable rolling updates. In the case of the database, the DB schema needs to be backward compatible to avoid disruption during rollback and loss of user data. In practice, it means columns can only be added (not removed), and column definition cannot be altered. Schema changes should be controlled with other change management frameworks, such as Flyway²¹, so DB changes can also follow the GitOps process.

3.5 Summary

- Environments are where code is deployed and executed for a specific purpose.
- Each environment will have its own access control, networking, configuration, and dependencies.
- Factors for picking environment granularity are release independence, test boundary, access control, and isolation.
- Kubernetes namespace is a natural construct to implement an environment.
- Since namespace is equivalent to an environment, deploying to a specific environment is merely specifying the targeted namespace.
- Inter-environment traffic can be controlled by network policy.
- Pre-prod and Prod should follow the same security best practices and operation vigor.
- Separation of Git repo for Kubernetes Manifest and Git repo for code is highly

²¹ <https://flywaydb.org/>

recommended to allow environment changes independent from Code changes.

- Single branch works well with tooling like Kustomize for overlay.
- Monorepo for Config works well for start-ups vs. Multiple repos works well for large enterprises.
- Helm is a package manager.
- Kustomize is a built-in config management tool part of `kubectl`.
- Jsonnet is a language that is for JSON templating.
- Choosing the right config management tools should base on the following criteria: declarative, readability, flexibility and maintainability.
- Durable environments are always on for others to use, and Ephemeral is for short-lived testing and previewing.

4

Pipelines

This chapter covers:

- What are the stages in a GitOps CI/CD pipeline?
- Promoting code, image, and environment
- Rollback
- Compliance pipeline

This chapter builds on the concepts learned in chapter 3 and discusses how pipelines are created to build and test application code and then deploy it to different environments. You will also learn about different promotion strategies and how to revert, reset, or rollback application changes.

We recommend you read chapters 1, 2, and 3 before reading this chapter.

4.1 Stages in CI/CD Pipelines

Continuous Integration (CI) is a software development practice in which all developers merge code changes in a central repository (Git). With CI, each code change (commit) triggers an automated build-and-test stage for the given repo and provides feedback to the developer(s) who made the change. The main difference between GitOps compared to traditional CI is that with GitOps, the CI pipeline also updates the application manifest with the new image version after the build and test stages have been completed successfully.

Continuous Delivery (CD) is the practice of automating the entire software release process. Continuous Delivery includes infrastructure provisioning in addition to deployment. What makes GitOps CD different from traditional CD is using a GitOps operator to monitor the manifest changes and orchestrate the deployment. As long as the CI build is complete and the manifest is updated, GitOps Operator takes care of the eventual deployment.

Please refer to section 2.5 for GitOps CI/CD and Operator basics.

©Manning Publications Co. To comment go to [liveBook](#)

Licensed to anand kumar kumar <akyhcs@gmail.com>

This chapter takes an in-depth look at a comprehensive CI/CD pipeline and why it is important for software development. A CI/CD pipeline is a collection of stages, and each stage performs a specific task to achieve the following objectives.

Productivity: Provide valuable feedback for the developers early in the development cycle in terms of design, coding style, and quality without context switching. Code Review, Unit Test, Code Coverage, Code Analysis, Integration Test, and Runtime vulnerability are essential stages for design, quality, and security feedback to the developers.

Security: Detect code and component vulnerabilities that are attack surfaces for potential exploitation. Vulnerability Scan can detect security issues with third-party libraries. Runtime Vulnerability Scan can detect runtime security issues with code.

Defect Escape: Reduce failure of customer interactions and costly rollback. A new release is typically providing new features or enhancing existing ones. If the features do not provide the correct functionality, the consequence will be customer dissatisfaction and potential revenue loss. Unit Tests verify correctness at the module level, and functional tests verify correctness across two or more modules.

Scalability: Discover scalability issues before the production release. Unit Tests and functional tests can verify the features correctness, but these stages cannot detect problems such as memory leak, thread leak, or resource contention issues. Canary Release is a way to deploy the new version to detect scalability issues using production traffic and dependencies.

Time to Market: Deliver features to customers quicker. With a fully automated CI/CD pipeline, there is no time-intensive manual work to “deploy” the software. The code can be released as soon as it passes all stages in the pipeline.

Reporting: Insight for continuous improvement and metrics for auditability. CI/CD pipeline execution time in minutes versus hours can affect developers’ behavior and productivity. Continuously monitoring and improving the pipeline execution time can dramatically improve team productivity. Collecting and storing Build metrics are also required for many regulatory audits. Please refer to the CI and CD metrics publishing stages for detail.

4.1.1 GitOps Continuous Integration

Figure 4.1 illustrates a comprehensive CI pipeline building on the GitOps CI/CD (Figure 2.9) in Chapter 2. The boxes in gray are new stages for a complete CI solution. This section will help you plan and design stages relevant to your business based on your complexity, maturity, and compliance requirements.

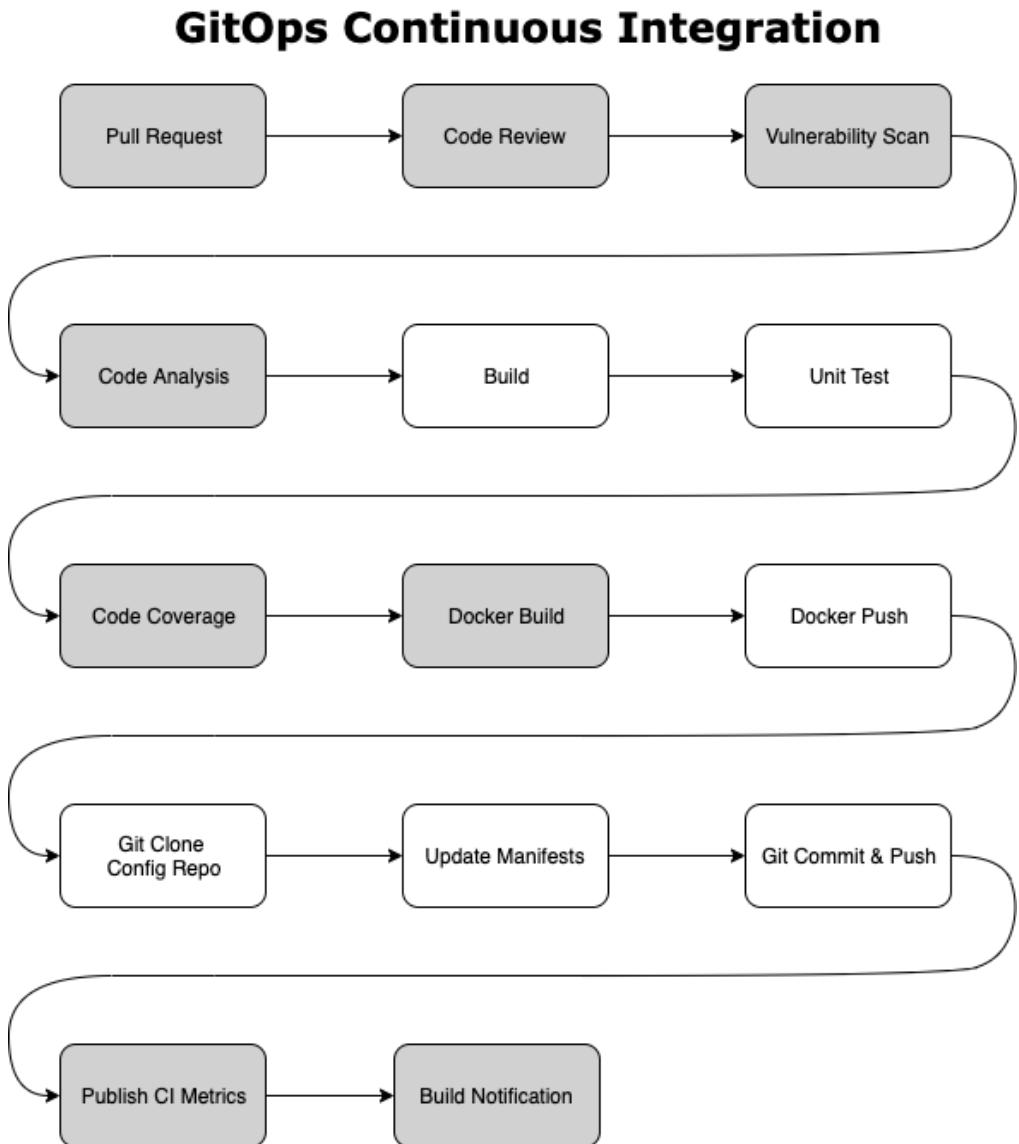


Figure 4.1 These are the stages in a GitOps CI pipeline. White boxes are from the GitOps CI pipeline from Figure 2.9, and Gray boxes are the additional stages for building a complete CI pipeline.

PRE-BUILD STAGES

The following stages are also known as *static analysis* stages. They are a combination of manual and automated scanning of the code before the code is built and packaged into a docker image.

Pull Request/Code Review

All CI/CD pipelines should always start with a Pull Request, which allows code review to ensure consistency between design and implementation and catch other potential errors. As discussed in chapter 1, Code Review also helps share best practices, coding standards, and team cohesion.

Vulnerability Scan

Open-source libraries can provide many functionalities without custom development, but those libraries can also come with vulnerabilities, defects, and licensing issues. Integrating an Open-source library scanning tool like Nexus Vulnerability Scanner can detect known vulnerabilities and licensing issues early in the development cycle and remediate the problems by either upgrading the libraries or using alternative libraries.

The old saying “it ain’t broken, don’t fix it” doesn’t work anymore in the rapidly changing software industry. Vulnerabilities are discovered every day with open source libraries, and it is prudent to upgrade as soon as possible to avoid being exposed to exploits. At Intuit, we leverage open-source software heavily to accelerate our development. Instead of doing an annual security audit, we now have a vulnerability scan step in our CI pipelines to detect and address security issues regularly during the development cycle.

Code Analysis

While manual code review is excellent for design and implementation consistency, coding standards, duplicate code, and code complexity issues (aka Code smell²²) are better suited using an automated linting or code analysis tool such as SonarQube. These tools are not a replacement for code review, but they can catch the “mundane” issues more effectively.

It is unrealistic to expect every minor issue to be fixed before the new code can be deployed. With tools such as SonarQube, the trend data is also reported so the team can see how their “Code Smell” is getting better or worse over time so the team can address these issues before they have gone too far.

Exercise 4.1

To prevent known security issues with your open-source library, what stage(s) do you need to plan in your CI/CD pipeline?

To ensure implementation matching the design, what stage(s) do you need to plan in your CI/CD pipeline?

²² https://en.wikipedia.org/wiki/Code_smell

BUILD STAGES

After static analysis, it is time to build the code. In addition to building and creating the deployable artifact (aka docker image), unit (module) testing and the effectiveness of the unit tests (code coverage) are also an integral part of the build process.

Build

The build stage typically starts with downloading the dependency libraries before the project source code's actual compilation. (Scripting languages such as Python or node.js do not require compilation.). For compiled languages like Java, Ruby, or Golang, the code is compiled into bytecode/machine binary using the respective compiler. Additionally, the generated binary and its dependency libraries need to be packaged into a deployable unit (ex: jar or war in Java) for deployment.

In our experience, the most time-consuming portion of the build is downloading the dependencies. It is highly recommended to cache your dependencies in your build system to reduce the build time.

Unit Test

A unit test is for verifying a small piece of code doing what it is supposed to do. Unit tests should have no dependencies on code outside the unit tested. Unit testing mainly focuses on testing the functionality of individual units only and does not uncover the issues that arise when different modules are interacting with each other. During unit testing, external calls are typically "mocked" to eliminate dependencies issues and reduce the test execution time.

In a unit test, mock objects can simulate the behavior of complex, real objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.²³ In our experience, Mocking is a "required" investment and will save the team time (faster test execution) and effort (troubleshooting flaky tests).

Code Coverage

Code coverage measures the percentage of code that is covered by automated unit tests. Code coverage measurement simply determines which statements in a body of code have been executed through a test run and which statements have not. In general, a code coverage system instruments the source code and gathers the runtime information to generate a report on the test suite's code coverage.

Code coverage is a critical part of a feedback loop in the development process. As tests are developed, code coverage highlights aspects of the code which may not be adequately tested and require additional testing. This loop continues until coverage meets some specified target. Coverage should follow an 80-20 rule as increasing coverage values becomes difficult and diminishes in return. Coverage measurement is not a replacement for thorough code review and programming best practices.

²³ https://en.wikipedia.org/wiki/Mock_object

Driving code coverage percentages higher alone can lead to wrong behavior and may actually lower quality. Code coverage measures the percentage of lines being executed but does not measure the correctness of the code. 100% code coverage with partial assertions will NOT achieve the quality goal of unit testing. Our recommendation is to focus on increasing the number of unit tests and code coverage over time instead of singly focused on an absolute code coverage number.

Docker Build

Docker image is the deployable unit for Kubernetes. Once code has been built, you can create the docker image with a unique **image id** for your build artifacts by creating a Dockerfile and executing docker build command. Docker image should have its unique naming convention, and each version should be “tagged” with a unique version number. Additionally, you can also run a Docker image scanning tool at this stage to detect potential vulnerability issues with your base images and dependencies.

DOCKER TAG AND GIT HASH: Since Git creates a unique hash for each commit, it is recommended to use the Git Hash to tag the Docker image instead of creating an arbitrary version number. In addition to uniqueness, each Docker image can easily trace back to the Git repo history using the Git hash to determine the exact code in the Docker image. Please refer to section 2.5.2 for additional information.

Docker Push

The newly built docker image needs to be published to a docker registry²⁴ for Kubernetes to orchestrate the eventual deployment. A Docker Registry is a stateless, highly scalable server-side application that stores and lets you distribute Docker images. For in-house development, the best practice is to host a private registry to have tight control over where the images are stored. Please refer to chapter 4 on how best to host a secured private docker registry.

Exercise 4.2

Plan the build stage(s) required so Code Coverage metrics can be measured.

If the image is tagged with the “latest” tag, can you tell what was packaged in the docker image?

GITOPS CI STAGES

With traditional CI, the pipeline will end after the build stages. With GitOps, additional GitOps specific stages are “required” to update the manifest for the eventual deployment.

Please refer to Figure 4.1.

Git Clone Config Repo

Assuming your Kubernetes config is stored in a separate repo, this stage performs “Git Clone” to clone the Kubernetes Config to the build environment for the subsequent stage to update your manifest.

²⁴ <https://docs.docker.com/registry/>

Update Manifests

Once you have the manifests in your build environment, you can update the manifests with the newly created image id using a configuration management tool, like kustomize. Depending on your deployment strategy, one or more environment-specific manifests are updated with the new image id. Please refer to section 3.3.2 for additional information on kustomize.

Git Commit and Push

Once the manifests are updated with a new image id, the last step is to commit the manifests back to the Git repo. The CI pipeline is complete at this point. Your GitOps Operator detects the change in your manifests and deploys the change to the Kubernetes cluster. The following is an example of implementing the three stages with the Git command.

There are two challenges with the GitOps stages, and exercise 4.4 will provide the steps to address these issues.

1. Which Git user should you use to track the manifest update and commit?
2. How do you handle concurrent CI builds which can update the repo simultaneously?

POST-BUILD STAGES

After everything is complete for the GitOps CI, additional stages are needed to gather metrics for continuous improvement and audit reporting and notify the team of the build status.

Publish CI metrics

CI metrics should be stored in a separate data store for:

- 1) Build Issues: Development teams need relevant data to triage issues with build failure or unit test failure.
- 2) Continuous Improvement: Long Build time can affect engineering teams' behavior and productivity. Reduction in Code coverage can potentially result in more production defects. Having historical build time and code coverage metrics enables teams to monitor trending, decrease build time, and increase code coverage.
- 3) Compliance requirements: For SOX2 or PCI requirements, build information such as test results, who did the release, and what was released are required anywhere from 14 months up to 7 years.

It is costly to maintain build history for greater than one year for most build systems. One alternative is to export the build metrics to external storage, such as S3, to fulfill the compliance and reporting requirements.

Build Notification

For CI/CD deployment, most teams would prefer the “no news is good news” model, which means that if all stages are successful, they don’t need to be bothered with the build status. In the case of build issues, teams should be informed right away so they can get feedback

and rectify the problem. This stage is typically implemented using team messaging or an email system, so teams can be notified as soon as the CI/CD pipeline is complete.

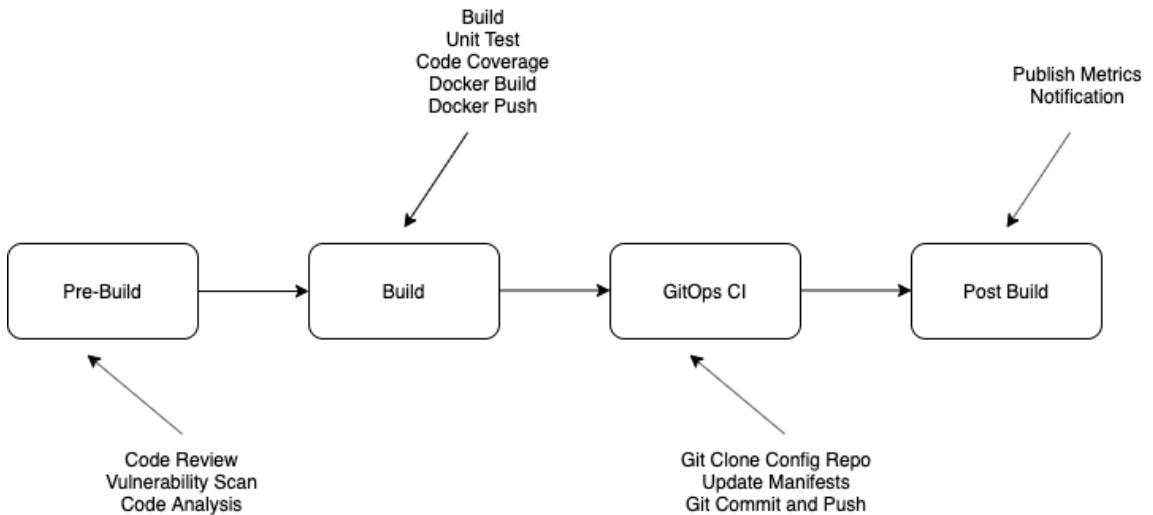


Figure 4.2 Pre-build will involve code review and static analysis. After build completion, GitOps CI will update the manifest (subsequently deployed by GitOps operator).

Exercise 4.3

What are the steps for the development team to determine build status if there is no build notification?

Is 80% code coverage good or bad? Hint: Trending

If the CI/CD pipeline typically takes an hour to run, what other tasks can developers do during that time? What if the CI/CD pipeline takes ten minutes instead?

Exercise 4.4

This exercise covers how to make Git changes for the new image id and addresses the challenges mentioned in the last section. Before you start, please fork the repo <https://github.com/gitopsbook/resources.git>. This exercise will assume your local computer is the build system.

- 1) Clone the repo from git. We will assume the guestbook.yaml under the folder chapter-04/exercise4.4 as your application manifest.

```
$ git clone https://github.com/<your repo>/resources.git
```

- 2) Use `git config` to specify the “committer” user email and name. Depending on your requirement, you can either use a service account or the actual committer account.

```
$ git config --global user.email <committerEmail>
$ git config --global user.name <commmitterName>
```

Please refer to chapter 4.2.1 GPG section for creating strong identity guarantees. The user-specified also needs to exist in your remote git repo.

- 3) Let’s assume the new docker image has git hashtag `zzzzzz`. We will update the manifest with tag `zzzzzz`.

```
$ sed -i .bak 's+acme.com/guestbook:.*$+acme.com/guestbook:zzzzzz+' chapter-04/exercise4.4/guestbook.yaml
```

To keep it simple, we will use `sed` to update the manifest in this exercise. Typically you should be using config tools like `kustomize` to update the image id.

- 4) Next, we will commit the change to the manifest.

```
$ git commit -am "update container for QAL during build zzzzzz"
```

- 5) Given the repo could be updated by others, we will run Git Rebase to pull down any new commit(s) to our local branch.

```
$ git pull --rebase https://<GIT_USERNAME>:<GIT_PASSWORD>@<your repo> master
```

- 6) Now we are ready to push the updated manifest back to the repo and let the GitOps operator do its deployment magic!

```
$ git push https://<GIT_USERNAME>:<GIT_PASSWORD>@<your repo> master
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 16 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 796 bytes | 796.00 KiB/s, done.
Total 7 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
remote: This repository moved. Please use the new location:
remote:   https://github.com/gitopsbook/resources.git
To https://github.com/gitops-k8s/resources
 eb1a692..70c141c master -> master
```

4.1.2 GitOps Continuous Delivery

Figure 4.3 illustrates a comprehensive CD pipeline building on the GitOps CI/CD (figure 2.9) in chapter 2. The boxes in gray are new stages for a complete CD solution. Depending on your complexity, maturity, and compliance requirements, you can pick and choose the relevant stages for your business.

The stages in the diagram depict the logical sequence. In practice, the GitOps stages are triggered by manifest changes in the Git repo and executed independently from the rest of the stages.

GitOps Continuous Delivery

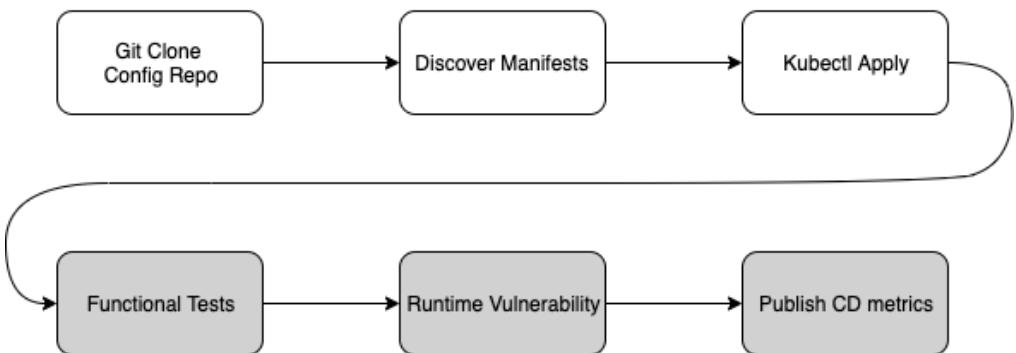


Figure 4.3 These are the stages in a GitOps CD pipeline. White boxes are from the GitOps CI pipeline from Figure 2.9, and Gray boxes are the additional stages for building a complete CI pipeline.

GITOPS CD STAGES

These are logical stages performed by the GitOps operator to deploy based on manifest change.

Git Clone Config Repo

The GitOps operator detects changes in your repo and performs a git clone to get your Git repo's latest manifests.

Discover Manifests

The GitOps Operator also determines any delta between the manifests in Kubernetes vs. the latest manifests from Git Repo. If there is no difference, GitOps Operator stops at this point.

Kubectl Apply

If the GitOps Operator determines differences between Kubernetes manifests vs. Git Repo manifests, GitOps Operator applies the new manifests to Kubernetes using the `kubectl apply` command.

Please refer to Figure 2.8 in section 2.5.2, poor-mans GitOps operator for detail.

POST DEPLOYMENT STAGES

After the image is deployed, we can now test the new code end to end against dependencies and runtime vulnerability.

Integration Tests

Integration testing is a type of testing to check if different modules are working correctly together. Once the image is deployed in the QA environment, integration testing can test across multiple modules and other external systems like databases and services. Integration testing aims to discover issues that arise when different modules interact to perform a higher-level function that cannot be covered by unit tests.

Since the GitOps operator handles the deployment outside of the pipeline, deployment might not be completed before the functional tests execution. Exercise 4.6 discusses the steps required to make integration tests working with GitOps CD.

Runtime Vulnerability

Runtime vulnerabilities are traditionally detected by penetration testing. Penetration testing, also called “pen testing” or ethical hacking, is the practice of testing a computer system, network, or web application to find security vulnerabilities that an attacker could exploit. Typically runtime vulnerabilities are SQL injection, command injection, or issuing “insecure” cookies. Instead of doing penetration testing in a production system (which is costly and after the fact), the QA environment can be instrumented using an agent tool like Contrast²⁵ while executing the integration tests to detect any runtime vulnerabilities early in the development cycle.

Publish CD metrics

CD metrics should be stored in a separate data store for:

- 1) Runtime Issue: Development teams need relevant data to triage issues with deployment, integration test failures, or runtime vulnerabilities.
- 2) Compliance requirements: For SOC2 or PCI requirements, build information such as test results, who did the release, and what was released are required anywhere from 14 months up to 7 years.

Exercise 4.5

Design a CD pipeline that can detect SQL injection vulnerability. Hint: SQL injection is a runtime vulnerability.

Exercise 4.6

This exercise covers how you can ensure changes are applied to Kubernetes, and the deployment completes successfully. We will use the `frontend-deployment.yaml` as our manifest.

Listing 4.1 `frontend-deployment.yaml`.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
```

²⁵ <https://www.contrastsecurity.com/>

```

name: frontend
labels:
  app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google-samples/gb-frontend:v4
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
          # service launched automatically. However, if the cluster you are using
          # does not have a built-in DNS service, you can instead
          # access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns' line above, and
          # uncomment the line below:
          # value: env
      ports:
        - containerPort: 80
  © 2020 GitHub, Inc.

```

- 1) Run “kubectl diff” to determine if the frontend-deployment.yaml manifest is applied to Kubernetes. “exit status 1” means manifest is NOT in Kubernetes.

```

$ kubectl diff -f frontend-deployment.yaml
diff -u -N /var/folders/s5/v3vpb73d6zv01dhxknw4yyxw0000gp/T/LIVE-
057767296/apps.v1.Deployment.gitops.frontend
/var/folders/s5/v3vpb73d6zv01dhxknw4yyxw0000gp/T/MERGED-
602990303/apps.v1.Deployment.gitops.frontend
--- /var/folders/s5/v3vpb73d6zv01dhxknw4yyxw0000gp/T/LIVE-
057767296/apps.v1.Deployment.gitops.frontend 2020-01-06 14:23:40.000000000 -0800
+++ /var/folders/s5/v3vpb73d6zv01dhxknw4yyxw0000gp/T/MERGED-
602990303/apps.v1.Deployment.gitops.frontend 2020-01-06 14:23:40.000000000 -0800
@@ -0,0 +1,53 @@
+apiVersion: apps/v1
+kind: Deployment
...
+status: {}
exit status 1

```

- 2) Apply the manifest to Kubernetes.

```
$ kubectl apply -f frontend-deployment.yaml
```

- 3) Rerun `kubectl diff`, and you should see the manifest is applied now with exit status 0. Kubernetes will start the deployment after the manifest update.

```
$ kubectl diff -f frontend-deployment.yaml
```

- 4) Repeatedly run `kubectl rollout status` until the deployment is fully completed.

```
$ kubectl rollout status deployment.v1.apps/frontend
Waiting for deployment "frontend" rollout to finish: 0 of 3 updated replicas are
available...
```

In production, you would automate this work using a script with a loop and sleep around the `kubectl rollout status` command.

Listing 4.2 DeploymentWait.sh.

```
#!/bin/bash
RETRY=0 #A
STATUS="kubectl rollout status deployment.v1.apps/frontend" #B
until $STATUS || [ $RETRY -eq 120 ]; do #C
    $STATUS #D
    RETRY=$((RETRY + 1)) #E
    sleep 10 #F
done

#A Initialize the RETRY variable to 0
#B Define the kubectl rollout status command
#C Loop exit condition on either kubectl rollout status is true or RETRY variable equals 120. This example
#will wait up to 20 minutes (120 x 10 seconds)
#D Execute the kubectl rollout status command
#E Increment the RETRY variable by 1
#F Sleep for 10 seconds
```

4.2 How to drive promotions

Now that we have covered all the stages in the CI/CD pipeline, we can take a look at how the CI/CD pipeline can automate the promotion of code, image, and environment. An automated environment promotion's main benefit is to enable your team to deploy new code to production faster and more reliably.

4.2.1 Code vs. Manifest vs. App Config

In section 3.2, we started the discussion on the Git strategy consideration for working with GitOps. We discussed the benefit of keeping code, and Kubernetes manifests in separate repos for more flexible deployment choices, better access control, and auditability. Where should we maintain application configuration for environment-specific dependencies such as database connection or distributed cache then? There are several options for maintaining the environment configuration.

Docker Image: All environment-specific App Config files can be bundled in the docker image. This strategy works best to quickly package legacy applications (with all environment app config bundled) into Kubernetes. The disadvantage is that creating a new environment requires a full build and cannot reuse existing images.

Config Map: ConfigMaps are native resources in Kubernetes and are stored in the Kubernetes etcd database. The disadvantage is that Pods need to be restarted if the ConfigMap is updated.

Config Repo: Storing App Config in a separate repo can achieve the same result as Config Map. The added benefit of this is the Pod can pick up changes in the application configuration dynamically (such as using Spring Cloud Config in Java).

Given an environment consisting of Code, Manifest, and App Config, any error in the repo changes could result in a production outage. Any changes to the Code, Manifest, or App Config repo should follow strict Pull Request/Code Review to ensure correctness.

Exercise 4.7

Assume the Code, Manifest, and App Config are kept in a single repo. You need to update the Manifest for one of the environment's replicas from X to Y. How can you commit the change only for GitOps deployment without building another image?

4.2.2 Code and Image promotion

Promotion is defined as “the act or fact of being raised in position or rank.” Code promotion means the code changes are committed to a feature branch and merged (promoted) with the master branch through a Pull Request. Once a new image is built and published by CI, it is then deployed (promoted) by the GitOps CD operator (as covered in section 4.1).

Imagine you are building a math library with addition and subtraction functions. You will first clone the master branch to create a new branch called Addition. Once you complete implementing the addition functionality, you commit the code to the Addition branch and generate a Pull Request to merge to the master branch. GitOps CI will create a new image and update the manifest. The GitOps operator will eventually deploy the new image. Then you can repeat the process to implement the subtraction functionality.

The code repo branching strategy has a direct impact on the image promotion process. Next, we discuss the pros and cons of a single- vs. multi-branching strategy on the image promotion process.

SINGLE BRANCH STRATEGY

Single Branch strategy is also called Feature Branch Workflow²⁶. In this strategy, the master branch is the official project history. Developers create short-lived feature branches for development. Once developers finish the feature, the changes are merged back to the

²⁶ <https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow>

master branch through the PR process. CI build will be triggered when the PR is approved to bundle the new code into a new docker image.

With single branch development, every image from the CI build can be promoted to any environment and used for a production release. If you need to roll back, you can re-deploy with any older images from the docker registry. This strategy is excellent and works best if your service can be deployed independently (aka microservice) and enables your team to do frequent production releases.

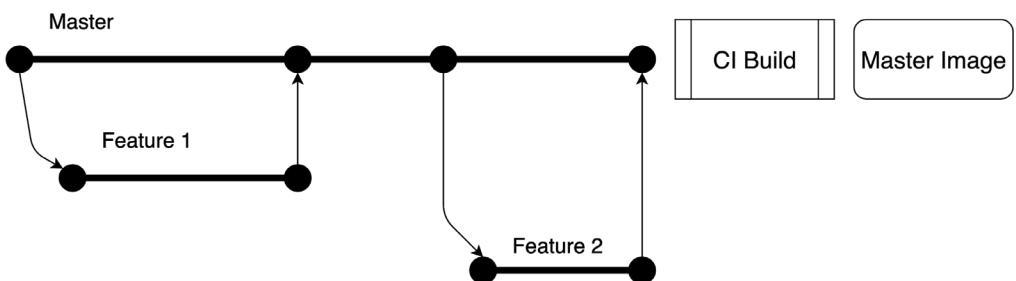


Figure 4.4 In the Single branch strategy, only the master branch is long-lived, and all feature branches will be short-lived. There is only one CI Build for the master branch.

MULTI-BRANCH STRATEGY

A *multi-branch strategy* is typically for larger projects which require close coordination of external dependencies and release planning. There are many variations of the Multi-Branch Strategy. For this discussion, we use the Gitflow Workflow²⁷ as an example. With Gitflow, the Develop branch has the official project history, and the master branch has the last production release history. The CI Build is configured for the Develop Branch for Continuous Integration. For feature development, developers create short-lived feature branches and merge changes to the Develop branch once the feature is complete.

When a release is planned, a short-lived release branch is forked from the latest development branch, and testing and bug fixing continues in this branch until the code is ready for production deployment. Hence, a separate CI build needs to be configured to build new docker images from the release branch. Once the release is complete, all changes are merged into the Develop and Master branches.

Unlike the single branch strategy, only the release CI build images can ever be deployed to production. All images from the Develop branch can only be used for pre-prod testing and integration. If rollback is needed, only images built from the release branch can be used. If roll-forward (or hotfix) is required for a production issue, a hotfix branch must be forked from the master branch, and a separate CI build created for the hotfix image.

²⁷ <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

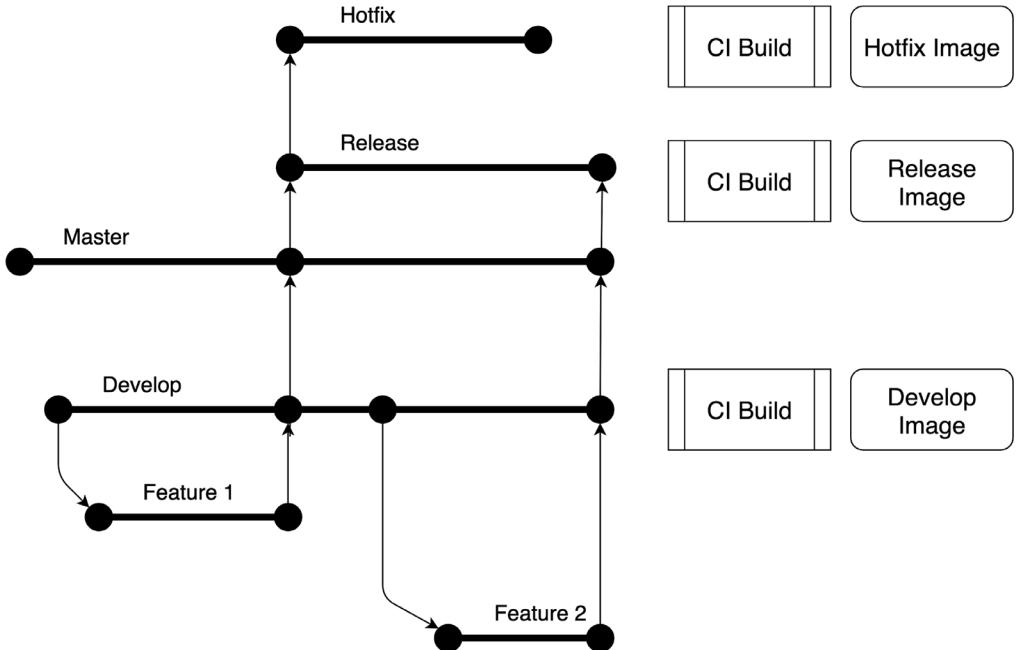


Figure 4.5 In the Multi-Branch strategy, there will be multiple long-lived branches, and each long-lived branch will have its own CI pipeline. In this example, the long-lived branches are Develop, Master, and Hotfix.

Exercise 4.8

Your service needs to release a feature on a specific date. Using a multi-branch strategy, design a successful release.

Using a single branch strategy, design a successful release for the specific date. Hint: feature flag

4.2.3 Environment promotion

In this section, we will discuss how to promote image from our pre-production to production environment. The reason for having multiple environments and promoting the change is to shift as much testing possible in the lower environment (shifting left), so we can detect and correct errors early in the development cycle.

There are two aspects to environment promotion. The first one is the environment infrastructure. As we discussed in chapter 3, Kustomize is the preferred config management tool to “promote” the new image to each environment, and the GitOps operator will do the rest!

The second aspect is the application itself. Since the docker image is the immutable binary, injecting the environment-specific app config will configure the application to behave for the specific environment.

In section 3.1, we introduced QA, E2E, Stage, and Production environments and discussed each environment's unique purpose in the development cycle. Let's review the stages that are important for each environment.

QA

QA environment is the first environment running the new image for verifying the correctness of the code during execution with external dependencies. The following stages discussed in section 4.1.3 are critical for the QA environment.

- 1) Functional Test
- 2) Runtime vulnerability
- 3) Publish metrics

E2E

E2E environment is primarily for other applications to test out existing or pre-release features. E2E environments should be monitored and operated similar to the production environment because E2E outage could potentially block the CI/CD pipelines of other services. An optional verification stage (sanity testing with the subset of the functional tests) is applicable for the E2E environment to ensure its correctness.

STAGE

Verify Production Dependencies

The Stage environment will typically connect to the production dependencies to ensure all production dependencies are in place before the production release. For example, a new version might depend on DB schema updates or message queues to be configured before it can be deployed. Testing with staging can guarantee all production dependencies are correct and avoid production issues.

PRODUCTION

Canary Release

"Canary release²⁸ is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody." We will have an in-depth discussion in Chapter 5 on Canary Release and how it can be implemented in Kubernetes.

Release Ticket

Given the complexity and distributed nature of application services, a release ticket is crucial for your production support team in case of a production incident. Release ticket will assist the production incident team in knowing what was deployed/changed by whom and what to roll back to if needed. Besides, release tracking is a must for compliance requirements.

²⁸ <https://martinfowler.com/bliki/CanaryRelease.html>

4.2.4 Putting it all together

This chapter started with defining the GitOps CI/CD pipeline to build a Docker image, verify the image, and deploy it to an environment. Then we discussed environment promotion with stages that are important for each environment. Figure 4.6 is an example of what a full Gitops CI/CD pipeline looks like with environment promotion.

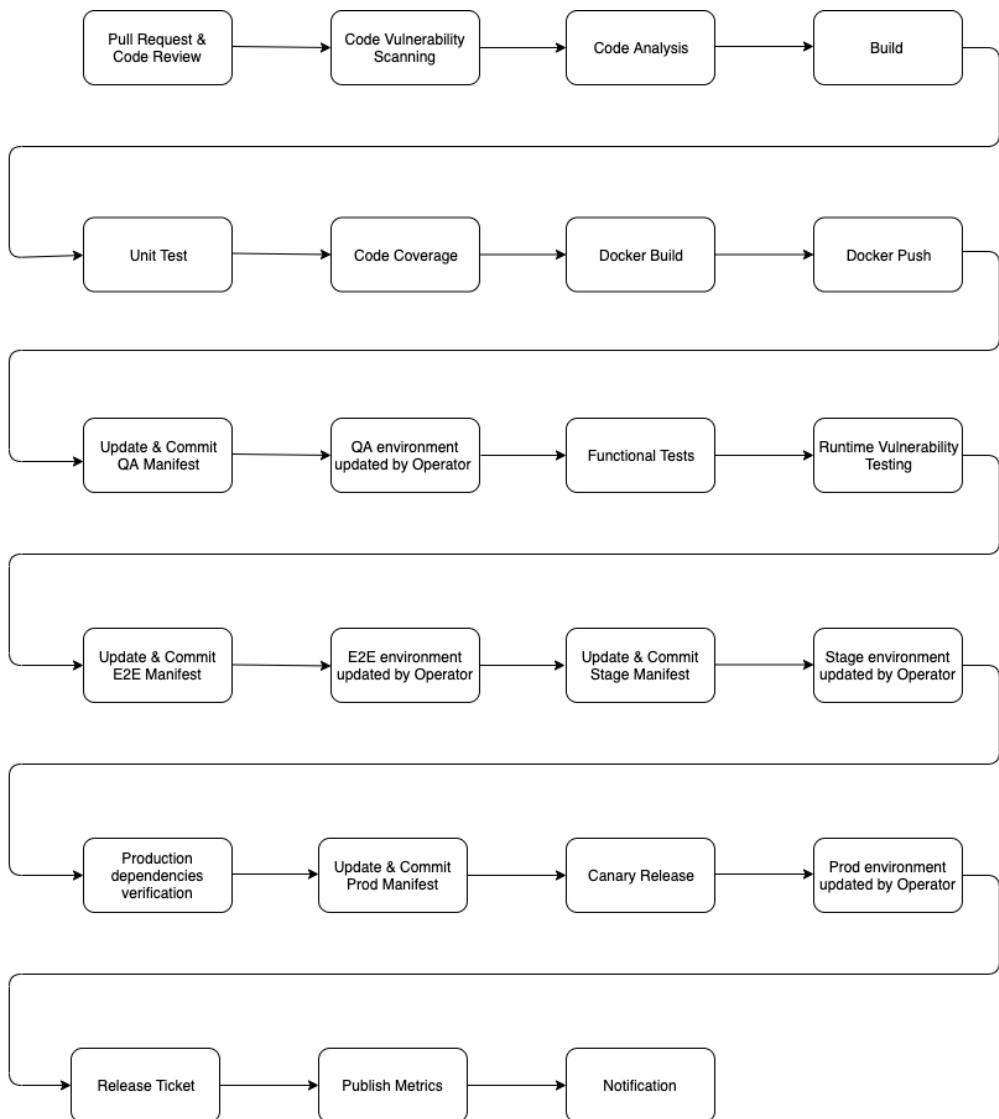


Figure 4.6 A CI/CD pipeline completes with environment promotion for single branch development. For multiple branch development, additional stages are required for branch promotion before environment

promotion.

SERIAL OR PARALLEL: Even though the diagram describes each stage as serial, many modern pipelines can support running stages in parallel. For example, notification and Metric publishing are mutually exclusive and can be executed in parallel to reduce the pipeline execution time.

4.3 Other pipelines

The CI/CD pipeline is primarily for your “happy path” deployment where your changes are working as expected, and life is good, but we all know that’s not the reality. From time to time, unexpected issues will arise in the production environment, and we will need to either rollback the environment or release “hotfixes” to mitigate the problem. With *SaaS* (Software as a Service), the highest priority is to recover from the production issues as quickly as possible and, in most cases, require a rollback to the previously known good state for a timely recovery.

For specific compliance standards, such as Payment Card Industry²⁹ (PCI), production releases require a second person to “approve” to ensure no single person can release changes to production. PCI also requires an annual audit that mandates reporting of the approval records. Given our original CI/CD pipeline will deploy changes to production per single PR, we need to enhance our pipeline to support compliance and auditability.

4.3.1 Rollback

Even if you have planned all the review, analysis, and testing stages in your CI/CD pipeline, eliminating all production issues is still impossible. Depending on the severity of the problem, you can either roll forward with fixes or rollback to restore your service to a previously known good state. Since our production environment consists of the manifest (which contains the Docker image id) and the application configuration for the environment, the rollback process could roll back the app config, manifest, or both repos. With GitOps, our rollback process is once again controlled by Git changes, and the GitOps operator will take care of the eventual deployment. (If the app config repo also needs to be rolled back, you just need to roll back the changes in App Config first before rolling back the manifest since only manifest changes can trigger a deployment, not App Config changes.) Git Revert and Git Reset³⁰ are two ways that can rollback changes in Git.

GIT REVERT: The git revert command can be considered an ‘undo’ command. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is essential for the integrity of your revision history (compliance and auditability) and for reliable collaboration. Please refer to the graphic at the top of figure 4.7 for an illustration.

²⁹ https://en.wikipedia.org/wiki/Payment_card_industry

³⁰ <https://www.atlassian.com/git/tutorials/undoing-changes/git-reset>

GIT RESET: This git reset command does a couple of different things, depending on how it is invoked. It modifies the index (the so-called "staging area"), or it changes which commit a branch head is currently pointing at. This command may alter existing history (by changing the commit that a branch references). Please refer to the graphic at the bottom of figure 4.7 for an illustration. Since this command can alter history, we do not recommend using git reset if compliance and auditability are important.

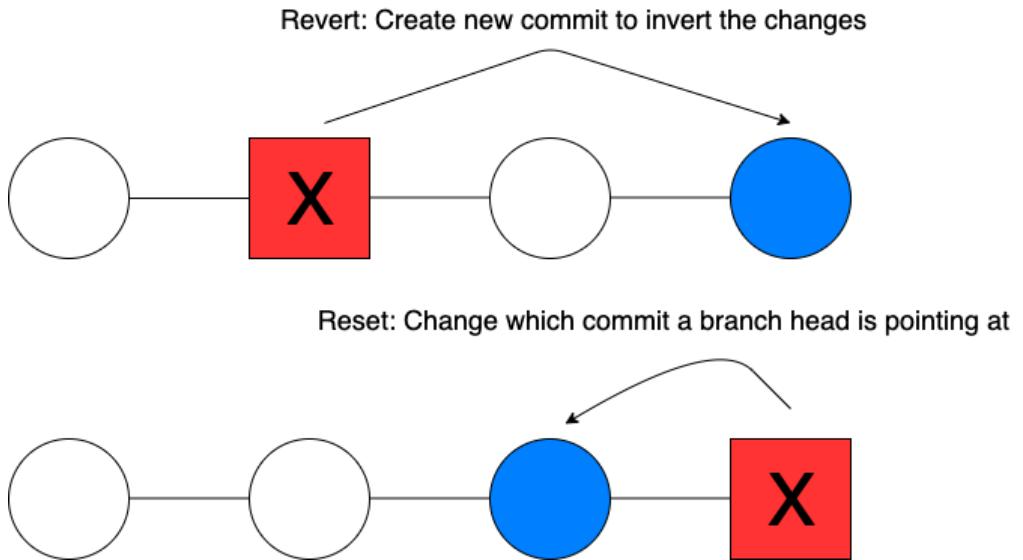


Figure 4.7 Git Revert works like an undo command with history preserved. Git Reset, on the other hand, modifies the history to “reset” the changes.

Figure 4.8 is an example of a rollback pipeline. This pipeline will start with Git Revert & Commit to rollback the manifest to the previously known good state. After a Pull Request is generated from the “revert” commit, approver(s) can approve and merge the PR to the manifest master branch. Once again, the GitOps operator will do its magic and rollback the application based on the updated manifest.

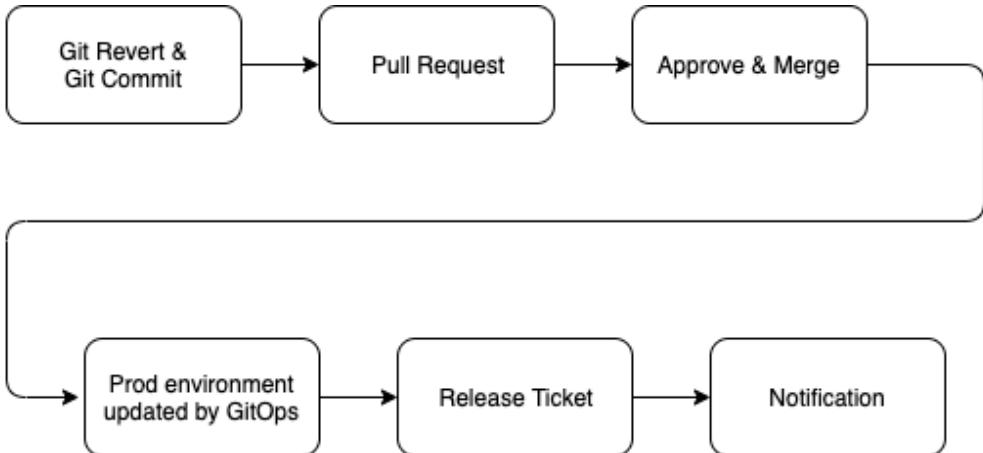


Figure 4.8 Rollback Pipeline involves reverting the manifest to a previous commit, generating a new Pull Request, and finally approving the PR.

Exercise 4.9

This exercise will go over the steps required to revert the image id from “zzzzzz” to “yyyyyy.” This exercise will use the “git revert” so the commit history is preserved. Before you start, please fork the repo <https://github.com/gitopsbook/resources.git>. This exercise will assume your local computer is the build system.

- 1) Clone the repo from git. We will assume the guestbook.yaml under the folder chapter-04/exercise4.9 as your application manifest.

```
$ git clone https://github.com/<your\_repo>/resources.git
```

- 2) Use git config to specify the “committer” user email and name. Depending on your requirement, you can either use a service account or the actual committer account.

```
$ git config --global user.email <committerEmail>
$ git config --global user.name <committerName>
```

Please refer to the GPG section 6.2.1 in chapter 6 for creating strong identity guarantees. The user that is specified also needs to exist in your remote git repo.

- 3) Let’s review the git history.

```
$ git log --pretty=oneline
eb1a692029a9f4e4ae65de8c11135c56ff235722 (HEAD -> master) guestbook with image hash
zzzzzz
95384207cbba2ce46ee2913c7ea51d0f4e958890 guestbook with image hash yyyyyy
4dcba52a809d99f9a1b5a24bde14116fad9a4ded (upstream/master, upstream/HEAD,
origin/master) exercise 4.6 and 4.10
```

```
e62161043d5a3360b89518fa755741c6be7fd2b3 exercise 4.6 and 4.10
74b172c7703b3b695c79f270d353dc625c4038ba guestbook for exercise 4.4
...
```

- 4) From the history, you will see "eb1a692029a9f4e4ae65de8c11135c56ff23572" for image hash zzzzzz. If we revert this commit, then the manifest will have an image hash yyyy. . .

```
$ git revert eb1a692029a9f4e4ae65de8c11135c56ff23572
```

- 5) Now we are ready to push the revert of the manifest by pushing back to the repo and let the GitOps operator do its deployment magic!

```
$ git push https://<GIT_USERNAME>:<GIT_PASSWORD>@<your repo> master
```

4.3.2 Compliance Pipeline

A Compliance pipeline essentially needs to ensure second person approval for production release and record by whom, when, and what gets released. In our case, we have created one CI/CD pipeline for pre-production development and a separate pipeline for production release. The pre-production CI/CD pipeline last stage will generate a PR to update the production manifest with the latest image id. When the approver wants to release a particular image to production, he/she can simply approve the respective PR and production environment will be updated by the GitOps Operator. Figure 4.9 illustrates stages for compliance Ci/CD and production release pipelines.

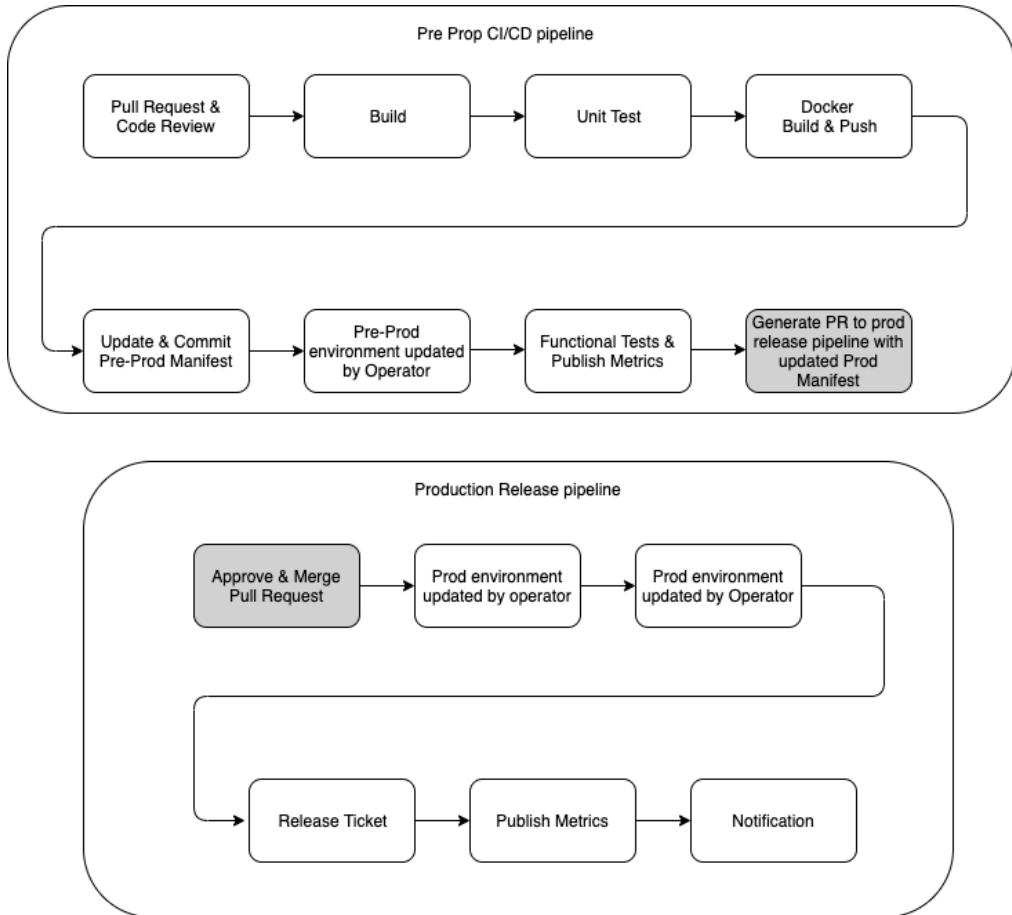


Figure 4.9 With the Compliance Pipeline, there is a separate production pipeline from the preprod CI/CD pipeline. At the end of the CI/CD pipeline, there is a stage to generate new PR with the new image id to the production manifest repository. Any PR getting approved will get deployed into the production.

Exercise 4.10

This exercise will create a new branch with your production manifest update and create a Pull Request back to the remote repo for approval. Before you start, please fork the repo <https://github.com/gitops-k8s/resources.git>. This exercise will assume your local computer is the build system.

- 1) Install the `hub`³¹ CLI for pull request creation. The hub CLI tool works with GitHub to fork branches and create pull requests.

³¹ <https://github.com/github/hub>

```
$ brew install hub
```

- 2) Clone the repo from git. We will assume the guestbook.yaml under the folder chapter-04 as your application manifest.

```
$ git clone https://github.com/<your\_repo>/resources.git
```

- 3) Use git config to specify the committer user email and name. Depending on your requirement, you can either use a service account or the actual committer account.

```
$ git config --global user.email <committerEmail>
$ git config --global user.name <commmitterName>
```

Please refer to chapter 4.2.1 GPG section for creating strong identity guarantees. The user specified also needs to exist in your remote git repo.

- 4) Create a new release branch.

```
$ git checkout -b release
```

- 5) Let's assume the new Docker image has the git hashtag "zzzzzz." We will update the manifest with the tag "zzzzzz."

```
$ sed -i .bak 's+acme.com/guestbook:.*$+acme.com/guestbook:zzzzzz+' chapter-04/exercise4.10/guestbook.yaml
```

- 6) Next, we will commit the change to the manifest.

```
$ git commit -am "update container for production during build zzzzzz"
```

- 7) Given the repo could be updated by others, we will run Git Rebase to pull down any new commit(s) to our local branch.

```
$ git pull --rebase https://<GIT_USERNAME>:<GIT_PASSWORD>@<your_repo> master
```

- 8) Fork the repo.

```
$ hub fork --remote-name=origin
```

- 9) Push the changes to your new remote.

```
$ git push https://<GIT_USERNAME>:<GIT_PASSWORD>@<your_repo> release
```

- 10) Open a pull request for the topic branch you've just pushed. This will also open up an editor for you to edit the pull request description. Once you save the description, this command will create the Pull Request.

```
$ hub pull-request -p
Branch 'release' set up to track remote branch 'release' from 'upstream'.
Everything up-to-date
```

- 11) Now you can go back to your remote repo, review, and approve the pull request in your browser!

4.4 Summary

- `git rebase` can mitigate conflict due to concurrent pipeline execution.
- Continuously running `kubectl rollout status` can ensure that deployment is complete and ready for running functional tests in GitOps CD.
- Having Code, Manifest, and App Config in separate repos will give you the best flexibility as infrastructure and code can evolve separately.
- The single branch strategy is great for smaller projects as every CI image can be promoted to production and zero branch management.
- A multi-branch strategy is excellent for larger projects with external dependencies and release planning. The downside is multiple long-lived branches must be maintained, and only release images can be deployed to production.
- A complete CI/CD pipeline will include environment promotion and stages for static analysis, build, unit/integration testing, and publishing build metrics/notification.
- Rollback of a production environment with GitOps is simply reverting the manifest to the previous commit (image id).
- GitOps pipelines naturally support compliance and auditability because all changes are generated as Pull Requests with approval and history.

5

Deployment Strategies

This chapter covers:

- Understanding why ReplicaSet is not a good fit for GitOps
- Understanding why Deployment is declarative and a good fit for GitOps
- Implementing Blue-Green Deployment using native Kubernetes resources and Argo Rollout
- Implementing Canary Deployment using native Kubernetes resources and Argo Rollout
- Implementing Progressive Delivery using Argo Rollout

In the previous chapters, we have focused on the initial deployment of Kubernetes resources. Launching a new application can be as simple as deploying a ReplicaSet with the desired number of Pod replicas and creating a Service to route the incoming traffic to the desired Pods. But now imagine you have hundreds (or thousands) of customers sending thousands of requests per second to your application? How do you safely deploy new versions of your application? How can you limit the damage if the latest version of your application contains a critical bug? In this chapter, you will learn about the mechanisms and techniques that can be used with Kubernetes to implement multiple different deployment strategies that are critical to running applications at enterprise- or internet-scale.

We recommend you read chapters 1, 2, and 3 before reading this chapter.

5.1 Deployment Basics

In Kubernetes, you can deploy a single pod using a manifest with PodSpec only. Suppose you want to deploy a set of identical pods with guaranteed availability. In that case, you can define a manifest with a ReplicaSet³² to maintain a stable set of replica Pods running at any given time. A ReplicaSet is defined with the selector that specifies how to identify Pods, the

³² <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>

number of replicas to be maintained, and a PodSpec. A ReplicaSet maintains the desired number of replicas by creating and deleting Pods as needed (Figure 5.1)

NOTE *ReplicaSet Is not declarative* ReplicaSet is not declarative and hence not suitable for GitOps. Section 5.1.1 will explain how ReplicaSet works and why it is NOT declarative in detail. Even though ReplicaSet is NOT declarative, it is still an important concept because the Deployment resource uses ReplicaSet objects to manage Pods.

Deployment³³ is a higher-level concept that leverages multiple ReplicaSets (Figure 5.1) to provide “declarative updates” to Pods along with a lot of other useful features (Section 5.1.2). Once you define the desired state in a Deployment manifest, the Deployment Controller will continue to observe the actual states and update the existing state to the desired state if they are different.

³³ <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

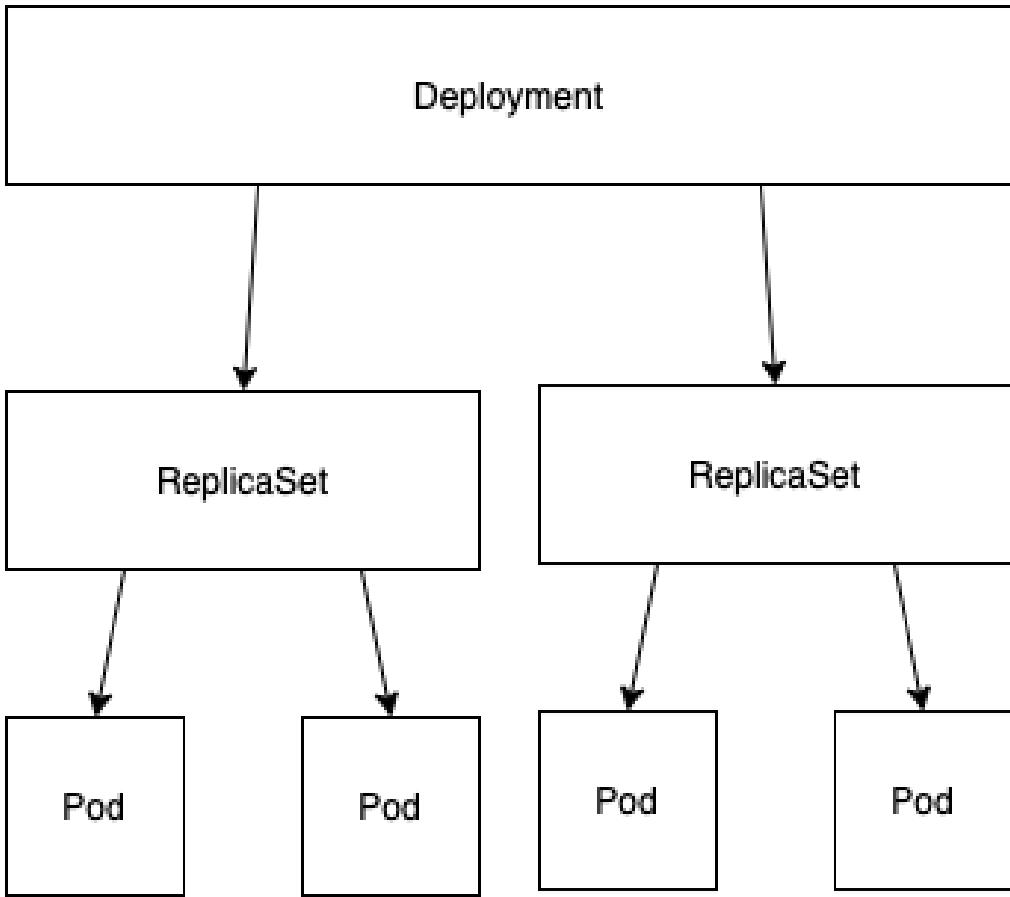


Figure 5.1 Deployments leverage one or more replicaSets to provide a declarative update of the application. Each replicaSet manages the actual number of pods based on podspec and the number of replicas.

5.1.1 Why ReplicaSet is NOT a good fit for GitOps

The ReplicaSet manifest includes a selector that specifies how to identify Pods it manages, the number of replicas of pods to maintain, and a pod template defining how new Pods should be created to meet the desired number of replicas. A ReplicaSet “Controller” will then create and delete Pods as needed to match the desired number specified in the manifest. As we mentioned early, ReplicaSet is not declarative, and we will use a tutorial to give you an in-depth understanding of how ReplicaSet works and why it is NOT declarative.

1. Deploy a ReplicaSet with two pods.
2. Update the image id in the manifest.
3. Apply the updated manifest and observe the ReplicaSet.
4. Update the replica to three in the manifest

5. Apply the updated manifest and observe the ReplicaSet.

If ReplicaSet is declarative, you should see three pods with the updated image id.

First, we will apply the replicaSet.yaml, which will create two pods with image id argoproj/rollouts-demo:blue and a service.

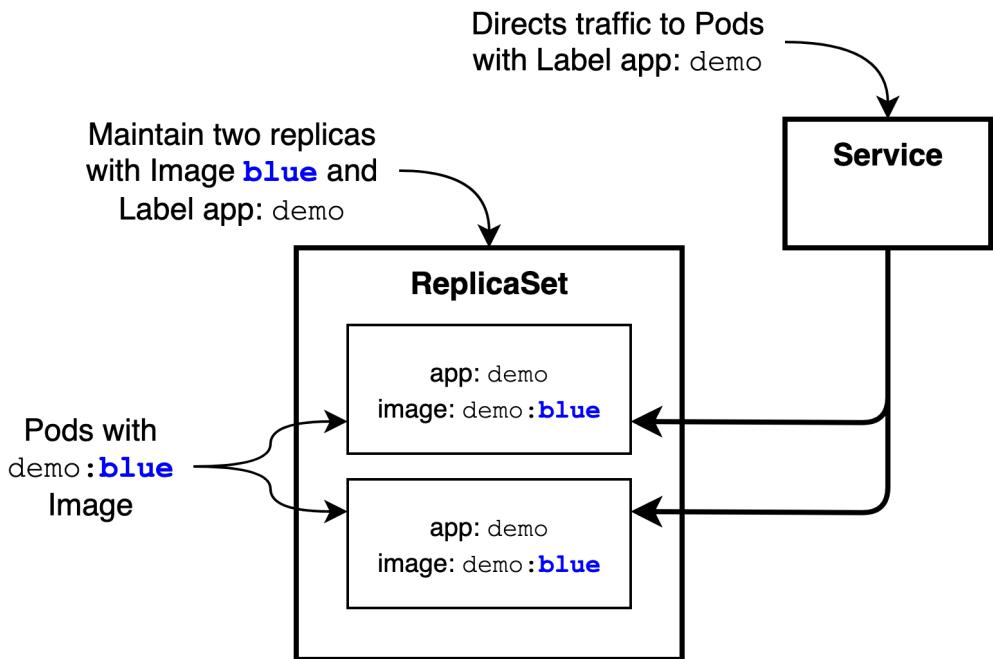


Figure 5.2 Applying the replicaSet.yaml will create two pods with image argoproj/rollouts-demo:blue. It also will create a demo service to direct traffic to the pods.

```
$ kubectl apply -f replicaSet.yaml
replicaset.apps/demo created
service/demo created
```

Listing 5.1 replicaSet.yaml.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 2    #A
  selector:
    matchLabels:
      app: demo
```

```

template:
  metadata:
    labels:
      app: demo
  spec:
    containers:
      - name: demo
        image: argoproj/rollouts-demo:blue    #B
        imagePullPolicy: Always
      ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: demo

```

#A Update replicas from 2 to 3

#B Update image tag from blue to green

After the deployment is completed, we will update the image id from blue to green and apply the changes.

```

$ sed -i .bak 's/blue/green/g' replicaSet.yaml
$ kubectl apply -f replicaSet.yaml
replicaset.apps/demo configured
service/demo unchanged

```

Next, we can use the `kubectl diff` command to verify the manifest has been updated in Kubernetes. Then we can run `kubectl get pods` and expect to see the image tag green instead of blue.

```

$ kubectl diff -f replicaSet.yaml
$ kubectl get pods -o jsonpath=".items[*].spec.containers[*].image"
argoproj/rollouts-demo:blue argoproj/rollouts-demo:blue

```

Even though the updated manifest has been applied, the existing pods did not get updated to green. Let's update the replica number from 2 to 3 and apply the manifest.

```

$ sed -i .bak 's/replicas: 2/replicas: 3/g' replicaSet.yaml
$ kubectl apply -f replicaSet.yaml
replicaset.apps/demo configured
service/demo unchanged
$ kubectl get pods -o jsonpath=".items[*].spec.containers[*].image"
argoproj/rollouts-demo:blue argoproj/rollouts-demo:green
argoproj/rollouts-demo:blue
$ kubectl describe rs demo
Name:           demo

```

```

Namespace: default
Selector: app=demo
Labels: app=demo
Annotations: kubectl.kubernetes.io/last-applied-configuration:
  {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations":{},"labels":{"app":"demo"},"name":"demo","namespace":"default"},"spe...
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels: app=demo
  Containers:
    demo:
      Image: argoproj/rollouts-demo:green
      Port: 8080/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
Events:
  Type Reason Age From Message
  ---- ---- - - -
  Normal SuccessfulCreate 13m replicaset-controller Created pod: demo-gfd8g
  Normal SuccessfulCreate 13m replicaset-controller Created pod: demo-gxl6j
  Normal SuccessfulCreate 10m replicaset-controller Created pod: demo-vbx9q

```

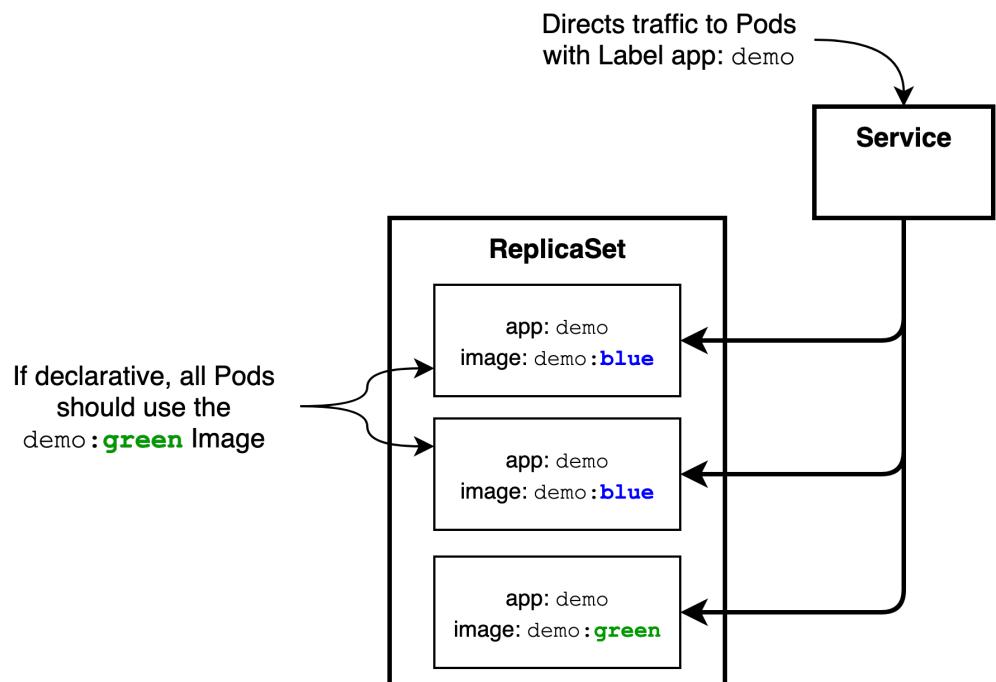


Figure 5.3 After applying the changes, you will see that only two pods are running the blue image. If ReplicaSet is declarative, all three pods should be green.

Surprisingly, the third pod's image tag is green, but the first two pods remain blue because Replica Controller's job is only to guarantee the number of running pods. If ReplicateSet were truly declarative, the ReplicaSet controller should detect the changes with image tag/replicas and update all three pods to green. In the next section, you will see how Deployment works and why it is declarative.

5.1.2 How Deployment works with ReplicaSets

Deployment is fully declarative and perfectly complementing GitOps. Deployment performs "Rolling Update" to deploy services with zero downtime. Let's go through a tutorial to examine how Deployment achieves "Rolling Update" using multiple ReplicaSets.

NOTE ***Rolling Update*** Rolling updates allow Deployments' update with zero downtime by incrementally updating Pods instances with new ones. Rolling update works great if your service is stateless and backward compatible. Otherwise, you will have to look into other deployment strategies, like Blue-Green, which will be covered in section 5.1.3

Let's imagine a real-life scenario of how this would be applicable. You run a Payments Service for small businesses to process credit cards. The service needs to be available 24x7, and you have been running two pods (Blue) to handle the current volume. You notice the two pods are maxing out, so you decide to scale up to three pods (Blue) to support the increased traffic. Next, your product manager likes to add debit card support, so you need to deploy a version with three pods (Green) with zero downtime.

1. Deploy two Credit Card (Blue) pods using Deployment
2. Review Deployment and ReplicaSet
3. Update replica from 2 to 3 and apply manifest
4. Review Deployment and ReplicaSet
5. Update the manifest with three Credit and Debit card (Green) Pods
6. Review Deployment and ReplicaSet while the three pods become "Green."

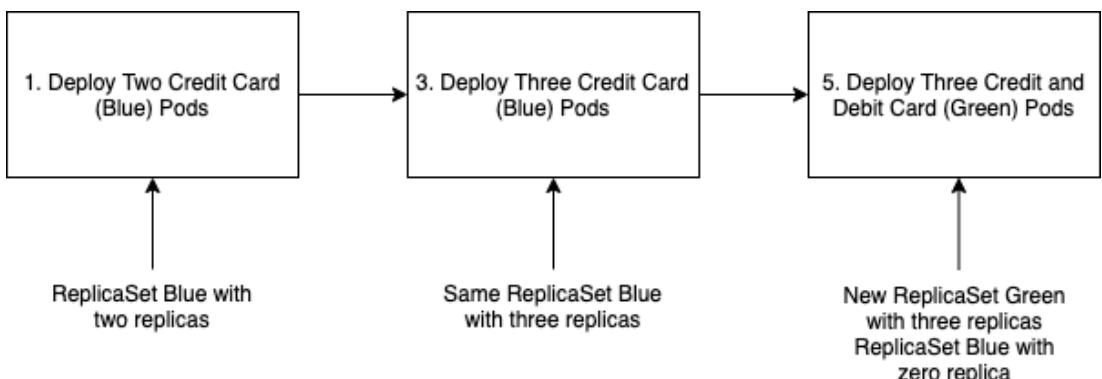


Figure 5.4 In this tutorial, you will initially deploy two blue pods. Then you would update/apply the manifest to three replicas. Finally, you will update/apply the manifest to three green pods.

Let's start with creating the initial Deployment. As you can see from Listing 5.2, the yaml is practically the same as Listing 5.1, with only changes to line 2 to use Deployment instead of ReplicaSet.

```
$ kubectl apply -f deployment.yaml
deployment.apps/demo created
service/demo created
```

Listing 5.2 deployment.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 2    #A
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: argoproj/rollouts-demo:blue    #B
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  Selector:
    app: demo           #C
```

#A replicas initially set to 2

#B Image initially tag set to blue

#C Service demo initially set to send traffic only to pods with Label app:demo

Let's review what was created after we applied the deployment manifest.

						#A
NAME	READY	STATUS	RESTARTS	AGE		
demo-8656dbfdc5-97slx	0/1	ContainerCreating	0	7s		
demo-8656dbfdc5-sbl6p	1/1	Running	0	7s		

```
$ kubectl get Deployment #B
NAME READY UP-TO-DATE AVAILABLE AGE
demo 2/2 2 2 61s

$ kubectl get rs #C
NAME DESIRED CURRENT READY AGE
demo-8656dbfdc5 2 2 2 44s

$ kubectl describe rs demo-8656dbfdc5 |grep Controlled #D
Controlled By: Deployment/demo

$ kubectl describe rs demo-8656dbfdc5 |grep Replicas #E
Replicas: 2 current / 2 desired

$ kubectl describe rs demo-8656dbfdc5 |grep Image #F
Image: argoproj/rollouts-demo:blue
```

#A Two running pods

#B One demo Deployment

#C One ReplicateSet demo-8656dbfdc5

#D The demo Deployment creates the ReplicateSet demo-8656dbfdc5.

#E ReplicateSet demo-8656dbfdc5 uses Image Id argoproj/rollouts-demo:blue.

As expected, we have one Deployment and one ReplicateSet demo-8656dbfdc5 created and controlled by Deployment demo. ReplicateSet demo-8656dbfdc5 manages two replicas of Pods with the blue image. Next, we will update the manifest with three replicas and review the changes.

```
$ sed -i .bak 's/replicas: 2/replicas: 3/g' deployment.yaml

$ kubectl apply -f deployment.yaml
deployment.apps/demo configured
service/demo unchanged

$ kubectl get pods
NAME READY STATUS RESTARTS AGE
demo-8656dbfdc5-97s1x 1/1 Running 0 98s
demo-8656dbfdc5-sbl6p 1/1 Running 0 98s
demo-8656dbfdc5-vh76b 1/1 Running 0 4s

$ kubectl get Deployment
NAME READY UP-TO-DATE AVAILABLE AGE
demo 3/3 3 3 109s

$ kubectl get rs
NAME DESIRED CURRENT READY AGE
demo-8656dbfdc5 3 3 3 109s

$ kubectl describe rs demo-8656dbfdc5 |grep Replicas
Replicas: 3 current / 3 desired

$ kubectl describe rs demo-8656dbfdc5 |grep Image
Image: argoproj/rollouts-demo:blue
```

After the update, we should see the same deployment and ReplicateSet that now manages the three blue pods. At this point, the deployment looks precisely as is depicted in figure 5.5.

Next, we will update the manifest to image “Green” and apply the changes. Since the image id has changed, Deployment will then create a second ReplicaSet to deploy the Green image.

NOTE Deployment and ReplicaSets A Deployment will create one ReplicaSet per image id and set the number of replicas to the desired value in the ReplicaSet with the matching image id. For all other ReplicaSets, Deployment will set those ReplicaSets’ replicas number to 0 to terminate all non-matching image id Pods.

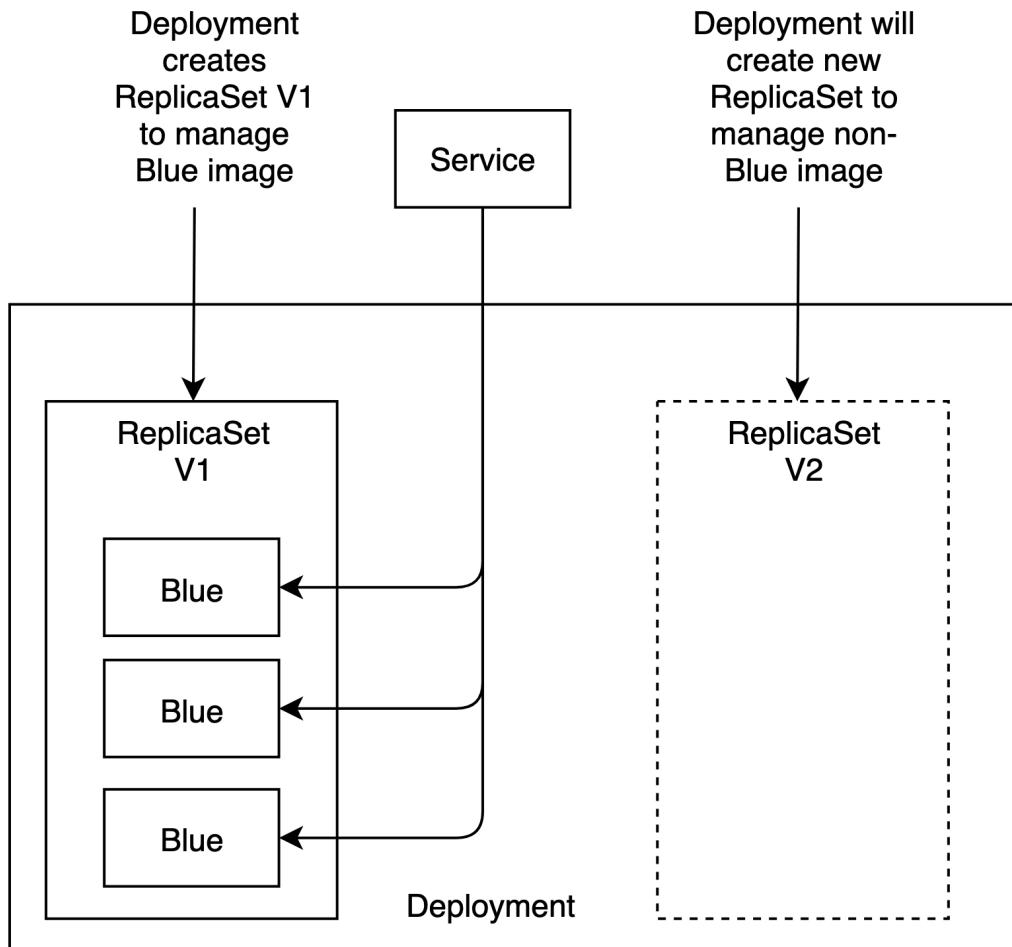


Figure 5.5 Deployment uses ReplicaSet V1 to maintain the “blue” deployment. If there is a change to the “non-blue” image, Deployment will create ReplicaSet V2 for the new deployment.

Before you apply the change, you can open up a new terminal with the following command to monitor the status of replicaSets. You should see one replica set (blue) with three pods.

```
$ kubectl get rs --watch
NAME      DESIRED   CURRENT   READY   AGE
demo-8656dbfdc5  3         3         3       60s
```

Go back to the original terminal, update the deployment and apply the changes.

```
$ sed -i .bak 's/blue/green/g' deployment.yaml
$ kubectl apply -f deployment.yaml
deployment.apps/demo configured
service/demo unchanged
```

Now switch to the terminal and you should see the replicaSet ‘demo-8656dbfdc5’ (blue) scaled down to 0 and new replicaSet ‘demo-6b574cb9dd’ (green) scaled up to 3.

```
$ kubectl get rs --watch
NAME      DESIRED   CURRENT   READY   AGE
demo-8656dbfdc5  3         3         3       60s          #A
demo-6b574cb9dd  1         0         0       0s           #B
demo-6b574cb9dd  1         0         0       0s
demo-6b574cb9dd  1         1         0       0s
demo-6b574cb9dd  1         1         1       3s
demo-8656dbfdc5  2         3         3       102s
demo-6b574cb9dd  2         1         1       3s
demo-8656dbfdc5  2         3         3       102s
demo-6b574cb9dd  2         1         1       3s
demo-8656dbfdc5  2         2         2       102s
demo-6b574cb9dd  2         2         1       3s
demo-6b574cb9dd  2         2         2       6s
demo-8656dbfdc5  1         2         2       105s
demo-8656dbfdc5  1         2         2       105s
demo-6b574cb9dd  3         2         2       6s
demo-6b574cb9dd  3         2         2       6s
demo-8656dbfdc5  1         1         1       105s
demo-6b574cb9dd  3         3         2       6s
demo-6b574cb9dd  3         3         3       9s          #C
demo-8656dbfdc5  0         1         1       108s
demo-8656dbfdc5  0         1         1       108s
demo-8656dbfdc5  0         0         0       108s          #D
```

#A Blue replicateSet began with three pods

#B Green replicateSet scaled up with one pod

#C Green replicateSet completed with three pods

#D Blue replicateSet completed zero pod

Let’s review what is happening here. Deployment uses the second ReplicaSet “demo-6b574cb9dd” to bring up one Green pod while uses the first ReplicaSet “demo-8656dbfdc5” to terminate one of the blue pods as depicted in Figure 5.6. This process will repeat until all three Green pods are created while all Blue pods are getting terminated.

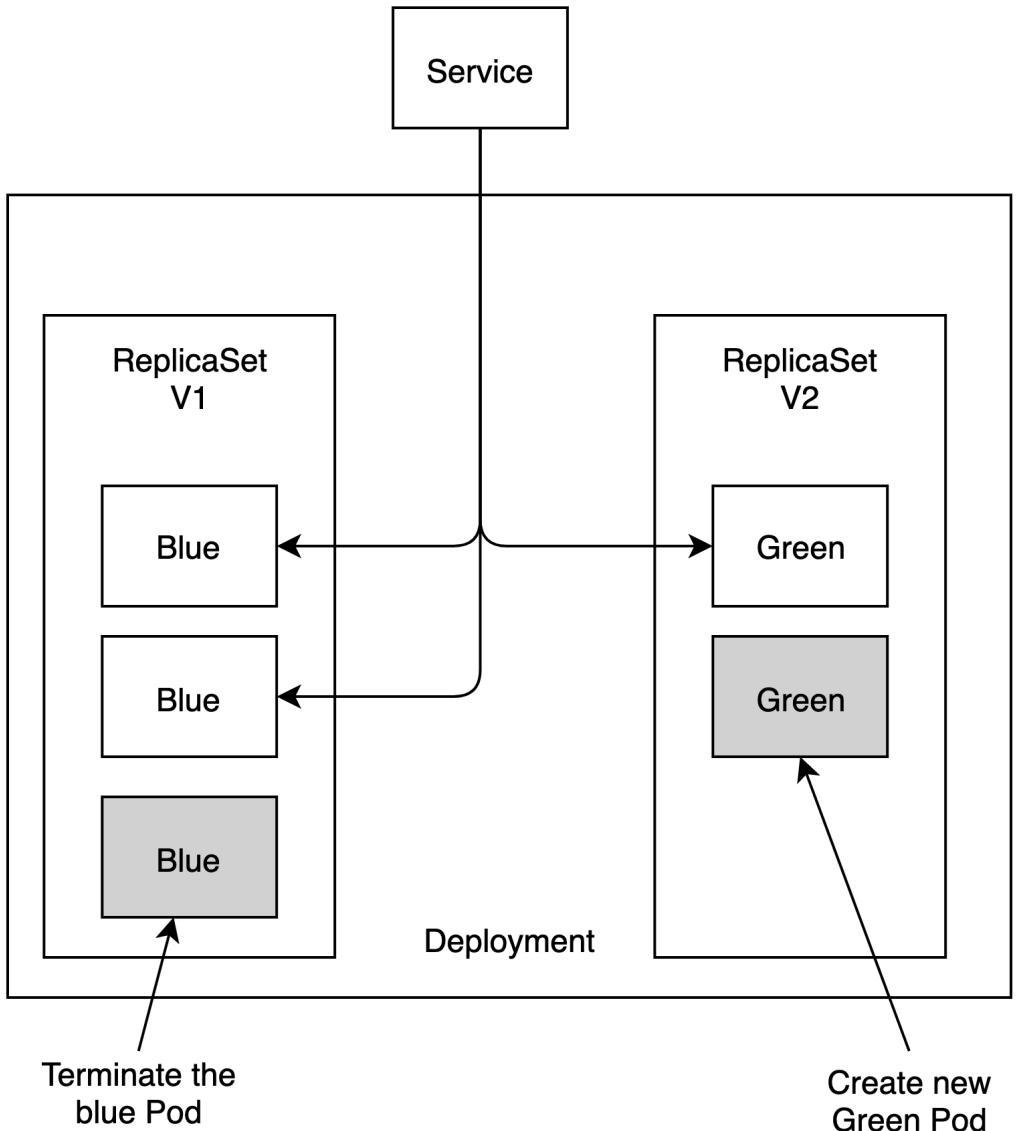


Figure 5.6 Deployment scales down ReplicaSet V1 while scales up ReplicaSet V2. ReplicaSet V1 will have zero pods, and ReplicaSet V2 will have three Green pods when the process is complete.

While we are discussing Deployment, we should also cover two important configuration parameters with the RollingUpdate strategy in Deployment: Max Unavailable and Max Surge. Let's review the default setting and what they mean according to the Kubernetes documentation.

```
$ kubectl describe Deployment demo |grep RollingUpdateStrategy
RollingUpdateStrategy: 25% max unavailable, 25% max surge
```

Deployment ensures that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable).

Deployment also ensures that only a certain number of Pods are created above the desired number of Pods. By default, it ensures that at most 125% of the desired number of Pods are up (25% max surge).

Let's see how it works in action. We will change the image id back to "Blue," configure the "Max Unavailable" to 3, and "Max Surge" to 3.

```
$ kubectl apply -f deployment2.yaml
deployment.apps/demo configured
service/demo unchanged
```

Now you can switch back to the terminal with replicaSet monitoring.

```
$ kubectl get rs --watch
NAME      DESIRED   CURRENT   READY    AGE
demo-8656dbfdc5  3         3         3        60s
demo-6b574cb9dd  1         0         0        0s
demo-6b574cb9dd  1         0         0        0s
demo-6b574cb9dd  1         1         0        0s
demo-6b574cb9dd  1         1         1        3s
demo-8656dbfdc5  2         3         3        102s
demo-6b574cb9dd  2         1         1        3s
demo-8656dbfdc5  2         3         3        102s
demo-6b574cb9dd  2         1         1        3s
demo-8656dbfdc5  2         2         2        102s
demo-6b574cb9dd  2         2         1        3s
demo-6b574cb9dd  2         2         2        6s
demo-8656dbfdc5  1         2         2        105s
demo-8656dbfdc5  1         2         2        105s
demo-6b574cb9dd  3         2         2        6s
demo-6b574cb9dd  3         2         2        6s
demo-8656dbfdc5  1         1         1        105s
demo-6b574cb9dd  3         3         2        6s
demo-6b574cb9dd  3         3         3        9s
demo-8656dbfdc5  0         1         1        108s
demo-8656dbfdc5  0         1         1        108s
demo-8656dbfdc5  0         0         0        108s
demo-8656dbfdc5  0         0         0        14m
demo-8656dbfdc5  3         0         0        14m
demo-6b574cb9dd  0         3         3        13m
demo-6b574cb9dd  0         3         3        13m
demo-8656dbfdc5  3         0         0        14m
demo-6b574cb9dd  0         0         0        13m #A
demo-8656dbfdc5  3         3         0        14m #B
demo-8656dbfdc5  3         3         1        14m
demo-8656dbfdc5  3         3         2        14m
demo-8656dbfdc5  3         3         3        14m
```

#A Green replicaSet immediately went to zero pod.

#B Blue replicaSet scaled up three pods at once.

As you can see from the ReplicaSet change status, you will notice that ReplicaSet `demo-8656dbfdc5` (green) immediately went to zero pods and ReplicaSet `demo-6b574cb9dd` (blue) immediately went to three instead of one at a time.

Listing 5.3 deployment2.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 3 #A
      maxUnavailable: 3 #B
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: argoproj/rollouts-demo:blue
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: demo #C
```

#A Create up to three pods at once.

#B Terminate up to three pods at once.

#C Service `demo` will send traffic to all pods with Label `app: demo`.

By now, you can see that Deployment achieves deployment with zero downtime by leveraging one ReplicateSet for “Blue” and another ReplicaSet for “Green.” As you learn about other Deployment strategies in the rest of the chapter, you will discover that they are all implemented similarly using two different ReplicaSets to achieve the desired motivation.

5.1.3 Traffic Routing

In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy to access them. The set of Pods targeted by a Service is determined by a selector that is a field within the Service manifest. Service will then forward traffic to pods with the matching labels as specified by the selector (also see listing 5.2 and 5.3). Service does round-robin load balancing and works great for Rolling Update if the underlying pods are stateless and backward compatible. If you need to customize load balancing for your deployment, you will need to explore other routing alternatives.

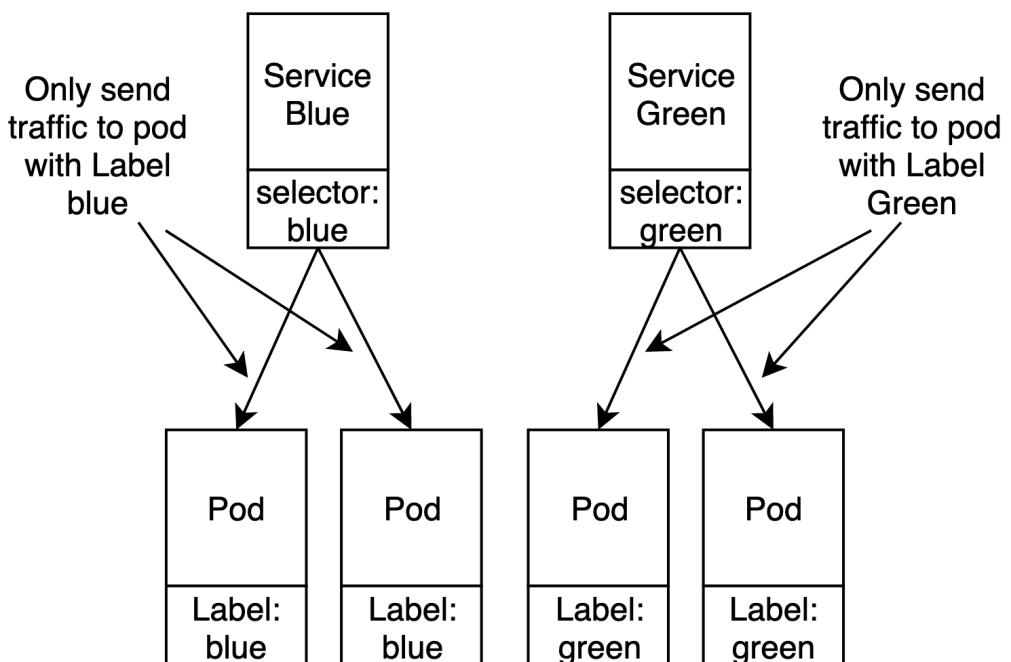


Figure 5.7 Service will only route traffic to pods with matching labels. In this example, Service Blue will only route traffic to Pods with Label Blue. Service Green will only route traffic to Pods with Label Green.

Nginx Ingress Controller³⁴: Nginx Ingress Controller can be used for many use-cases and supports various balancing and routing rules. Ingress Controller can be configured as the front load balancer to perform custom routing such as TLS termination, URL rewrite, or routing traffic to any number of services by defining the custom rules. Figure 5.8 illustrates the nginx ingress configured with Rules to send 40% incoming traffic to Service Blue and 60% incoming traffic to Service Green.

³⁴ <https://kubernetes.github.io/ingress-nginx/user-guide/basic-usage/>

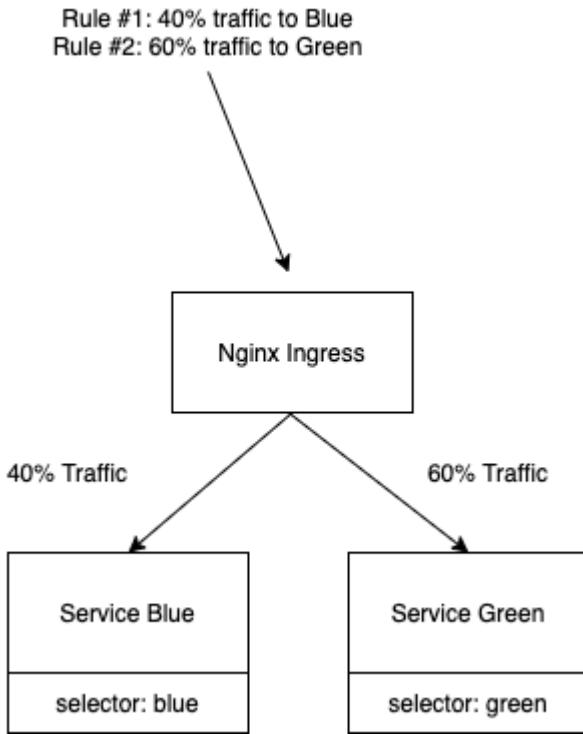


Figure 5.8 Nginx Ingress Controller can provide advanced traffic control. In this example, the nginx ingress controller is configured with one rule to send 40% traffic to Service Blue and a second rule to send 60% traffic to Service Green.

Istio Ingress Gateway³⁵: Istio Gateway is a load balancer operating at the Kubernetes cluster's edge, receiving incoming or outgoing HTTP/TCP connections. The specification describes a set of ports that should be exposed, the type of protocol to use, and custom routing configuration. The Istio Gateway will direct the incoming traffic to the back services (40% Service Blue, 60% Service Green) based on the custom configuration (Figure 5.9).

³⁵ <https://istio.io/docs/reference/config/networking/gateway/>

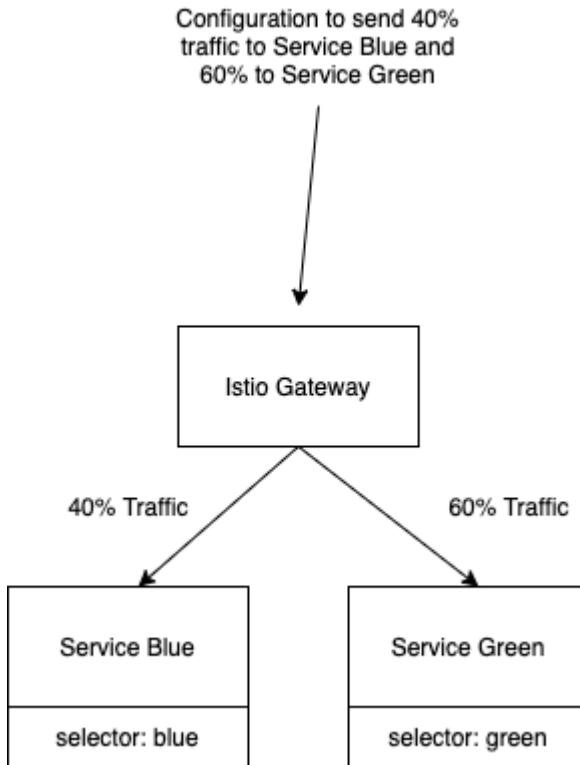


Figure 5.9 Istio Gateway is another load balancer that supports rich configuration for traffic routing. A custom configuration is defined to send 40% traffic to Service Blue and 60% traffic to Service Green in this example.

NOTE Both Nginx Ingress Controller and Istio Ingress Gateway are advanced topics and beyond the scope of this book. Please refer to the links in the footnote for additional information.

5.1.4 Configure minikube for other strategies

For the rest of the tutorial, you will need to enable nginx ingress and Argo Rollouts³⁶ support in your Kubernetes cluster.

NOTE Argo Rollouts The Argo Rollouts controller uses the Rollout custom resource to provide additional deployment strategies such as Blue-Green and Canary to Kubernetes. The Rollout custom resource provides feature parity with the deployment resource but with additional deployment strategies.

With minikube³⁷, you can enable nginx ingress support simply by running the following command.

³⁶ <https://github.com/argoproj/argo-rollouts>

```
$ minikube addons enable ingress
🌟 The 'ingress' addon is enabled
```

To install Argo Rollouts in your cluster, you just need to create an argo-rollouts namespace and run the install.yaml. For other environments, please refer to the Argo Rollouts Getting started guide³⁸.

```
$ kubectl create ns argo-rollouts
namespace/argo-rollouts created
$ kubectl apply -n argo-rollouts -f https://raw.githubusercontent.com/argoproj/argo-rollouts/stable/manifests/install.yaml
customresourcedefinition.apiextensions.k8s.io/analysisruns.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/analysistemplates.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/experiments.argoproj.io created
customresourcedefinition.apiextensions.k8s.io/rollouts.argoproj.io created
serviceaccount/argo-rollouts created
role.rbac.authorization.k8s.io/argo-rollouts-role created
clusterrole.rbac.authorization.k8s.io/argo-rollouts-aggregate-to-admin created
clusterrole.rbac.authorization.k8s.io/argo-rollouts-aggregate-to-edit created
clusterrole.rbac.authorization.k8s.io/argo-rollouts-aggregate-to-view created
clusterrole.rbac.authorization.k8s.io/argo-rollouts-clusterrole created
rolebinding.rbac.authorization.k8s.io/argo-rollouts-role-binding created
clusterrolebinding.rbac.authorization.k8s.io/argo-rollouts-clusterrolebinding created
service/argo-rollouts-metrics created
deployment.apps/argo-rollouts created
```

5.2 Blue-Green

As you learned in Section 5.1, Deployment's Rolling update is a great way to update applications because your app will use about the same amount of resources during a deployment with zero downtime and minimal impact on performance. However, there are many legacy applications out there that don't work well with rolling updates due to backward incompatibility or statefulness. Some applications may also require deploying a new version and cutting over to it right away or fast rollback in case of issues.

For these use cases, a blue-green deployment will be the appropriate deployment strategy. Blue-green deployment accomplishes these motivations by having two Deployments fully scale at the same time, but only direct coming traffic to one of the two deployments.

NOTE In this tutorial, we will use the nginx ingress controller to route 100% traffic to either blue or green deployment since the built-in Kubernetes Service³⁹ only manipulates the iptables⁴⁰ and does not reset the existing connection to the pods, therefore not suitable for blue-green deployment.

³⁷ <https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/>

³⁸ <https://argoproj.github.io/argo-rollouts/getting-started/>

³⁹ <https://kubernetes.io/docs/concepts/services-networking/service/>

⁴⁰ <https://en.wikipedia.org/wiki/Iptables>

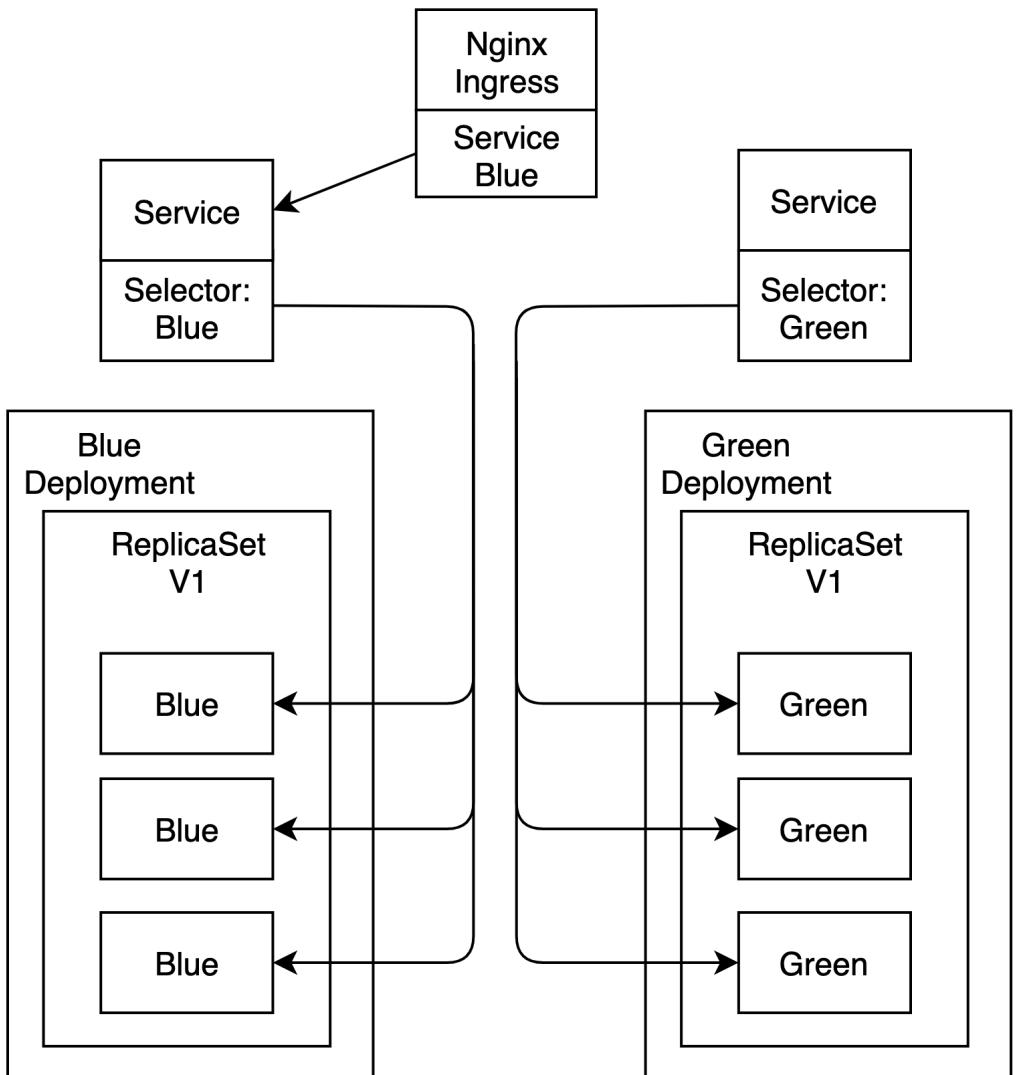


Figure 5.10 The initial deployment will configure the nginx ingress controller to send all traffic to the Service Blue. Service Blue will in turn send traffics to the Blue Pods.

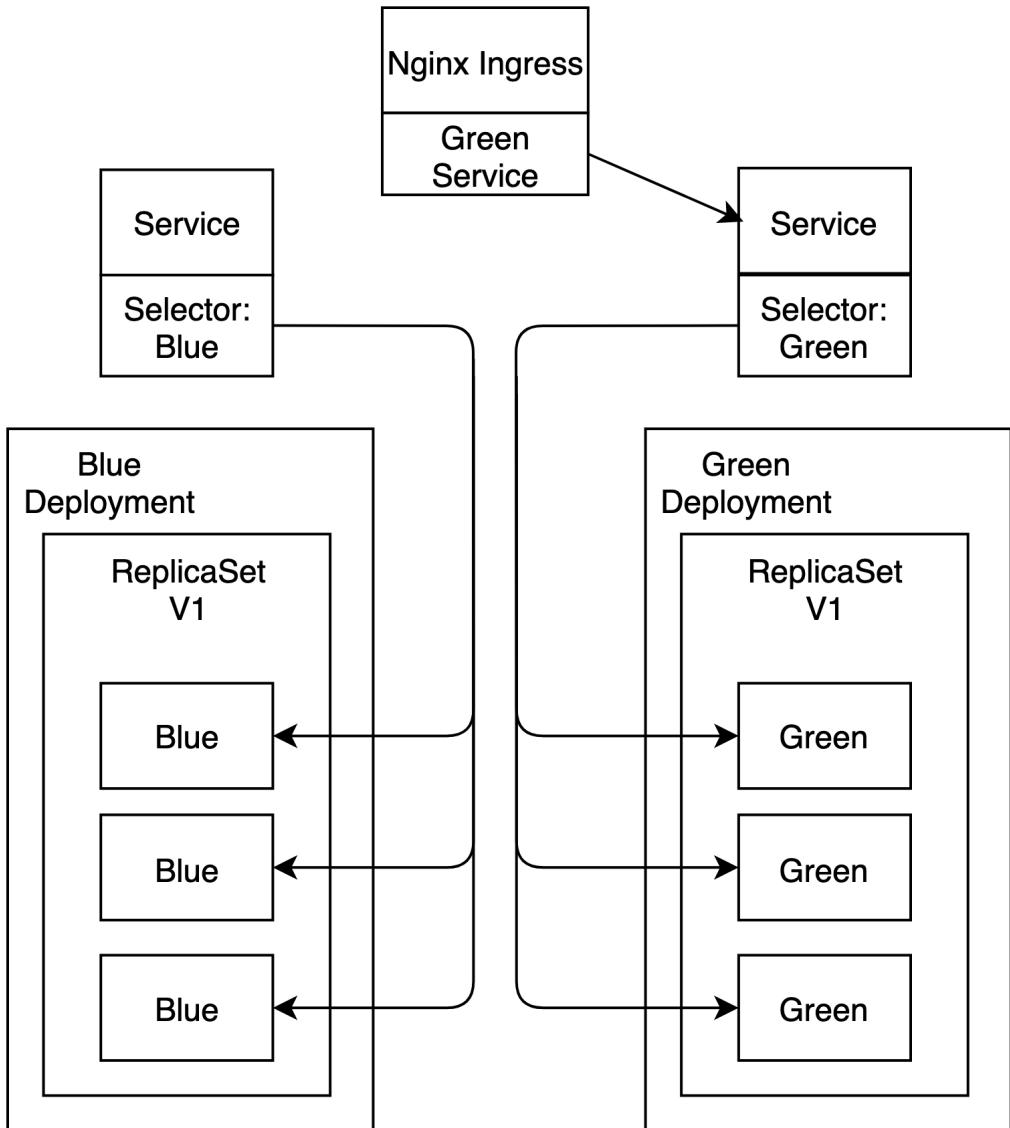


Figure 5.11 After the configuration update in the nginx ingress controller, all traffic will be sent to Service Green. Service Green will in turn send traffics to the Green Pods.

5.2.1 Blue-Green with Deployment

In this tutorial, we will perform a Blue-Green Deployment using native Kubernetes Deployment and Service.

NOTE Please refer to section 5.1.4 on how to enable ingress and install Argo Rollouts in your Kubernetes cluster prior to this tutorial.

1. Create a Blue Deployment and Service.
2. Create ingress to direct traffic to blue service.
3. View the application in the browser (blue).
4. Deploy Green Deployment and Service and wait for all pods to be ready.
5. Update ingress to direct traffic to green service.
6. View the web page again in the browser (green).

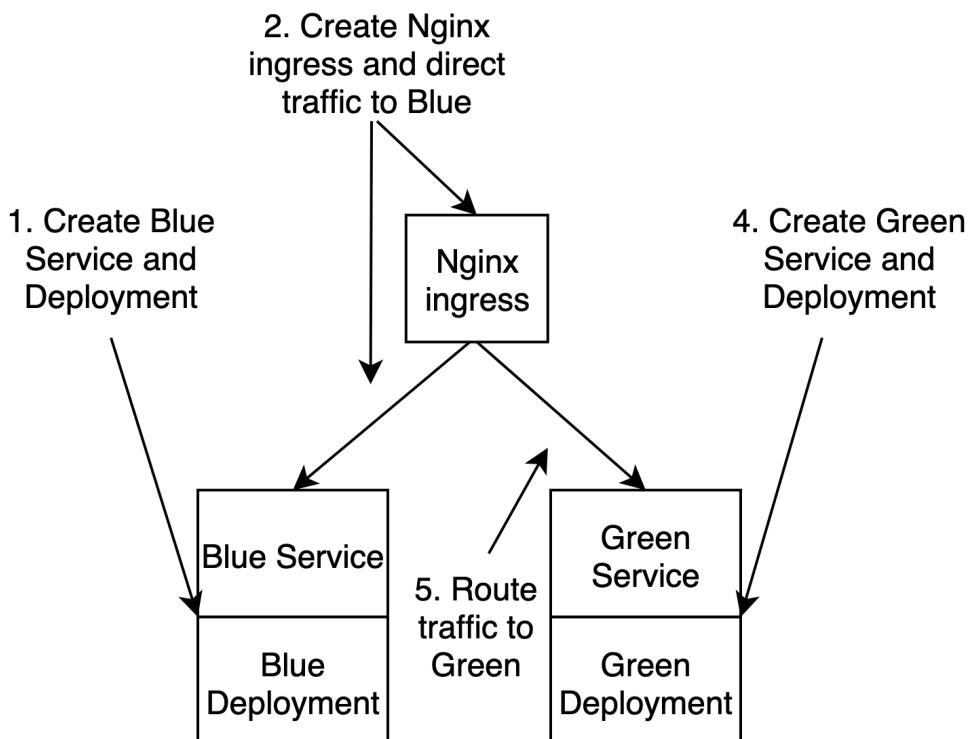


Figure 5.12 The steps for Blue-Green Deployment with Nginx Ingress are creating the initial state with Blue Service and Deployment along with the Nginx Ingress controller directing traffic to Blue Service. Then create the Green Service and Deployment with configuration change to Nginx Ingress Controller to direct traffic to Green Service.

We will first create the blue deployment by applying the `blue_deployment.yaml`.

```
$ kubectl apply -f blue_deployment.yaml
deployment.apps/blue created
service/blue-service created
```

Listing 5.4 blue_deployment.yaml.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: blue
  labels:
    app: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: blue
  template:
    metadata:
      labels:
        app: blue
    spec:
      containers:
        - name: demo
          image: argoproj/rollouts-demo:blue
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: blue-service
  labels:
    app: blue
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
    selector:
      app: blue
  type: NodePort

```

Now we can expose an ingress controller, so the `blue` service is accessible from your browser by applying the `blue_ingress.yaml`. The `kubectl get ingress` command will return the ingress controller hostname and IP address.

```

$ kubectl apply -f blue_ingress.yaml
ingress.extensions/demo-ingress created
configmap/nginx-configuration created
$ kubectl get ingress
NAME      HOSTS      ADDRESS      PORTS      AGE
demo-ingress  demo.info  192.168.99.111  80      60s

```

NOTE The nginx ingress controller will only intercept traffic with the hostname defined in the custom rule. Please make sure you add “`demo.info`” and its IP address to your `/etc/hosts`.

Listing 5.5 blue_ingress.yaml.

```
apiVersion: extensions/v1beta1
```

```

kind: Ingress
metadata:
  name: demo-ingress
spec:
  rules:
    - host: demo.info          #A
      http:
        paths:
          - path: /             #B
            backend:
              serviceName: blue-service #C
              servicePort: 80
---
apiVersion: v1
kind: ConfigMap           #D
metadata:
  name: nginx-configuration
data:
  allow-backend-server-header: "true"   #E
  use-forwarded-headers: "true"         #F

```

#A Assign the hostname ‘demo.info’ for the ingress controller

#B Route all traffic

#C Route traffic to blue-service at port 80

#D ConfigMap for customizing the headers control in Nginx

#E Enables the return of the header Server from the backend instead of the generic nginx string

#F Passes the incoming X-Forwarded-* headers to upstreams

Once you have the ingress controller, blue service, and deployment created and updated /etc/hosts with “demo.info” and the correct IP address, you can enter the url `demo.info` and see the blue service running.

NOTE The demo app will continue to call the “active” service in the background and show the latest results on the right side. Blue (darker gray in print) is the running version, and Green (lighter gray in print) is the new version.

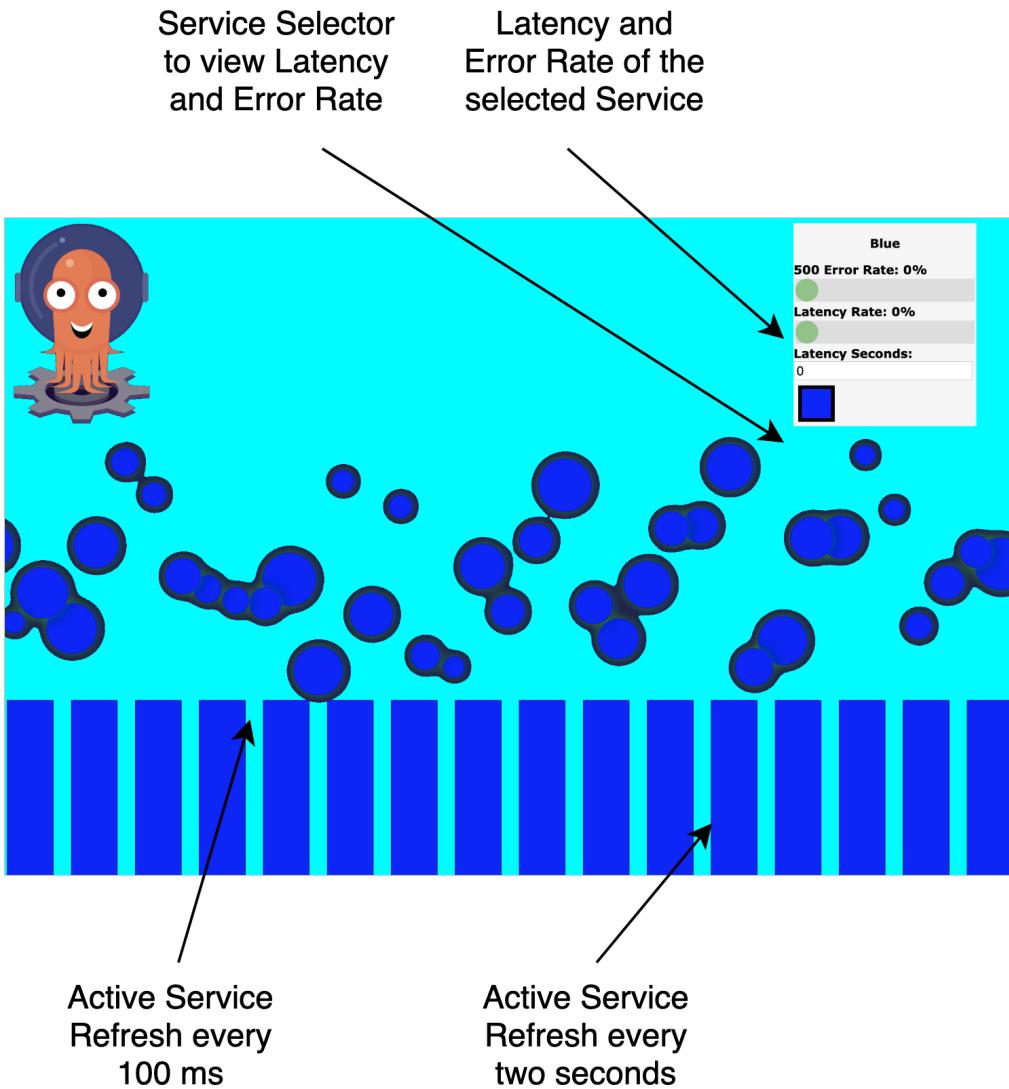


Figure 5.13 The HTML page will refresh every two seconds in the bar chart and every 100ms in the “bubble” chart to show either blue or green service responding. Initially the HTML page will show all blue because only all traffic are going to Blue Deployment.

Now we are ready to deploy the new Green version. Let's apply the `green_deployment.yaml` to create the Green service and deployment.

```
$ kubectl apply -f green_deployment.yaml
deployment.apps/green created
service/green-service created
```

Listing 5.6 green_deployment.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: green
  labels:
    app: green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: green
  template:
    metadata:
      labels:
        app: green
    spec:
      containers:
        - name: green
          image: argoproj/rollouts-demo:green
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: green-service
  labels:
    app: green
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: green
  type: NodePort
```

With the green service and deployment ready, we can now update the ingress controller to route traffic to the green service.

```
$ kubectl apply -f green_ingress.yaml
ingress.extensions/demo-ingress configured
configmap/nginx-configuration unchanged
```

Listing 5.7 green_ingress.yaml.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
```

```

rules:
- host: demo.info
  http:
    paths:
      - path: /
        backend:
          serviceName: green-service      #A
          servicePort: 80
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-configuration
data:
  allow-backend-server-header: "true"
  use-forwarded-headers: "true"

```

#A Route traffic to green-service instead of blue-service.

If you go back to your browser, you should see the service turning “Green”!

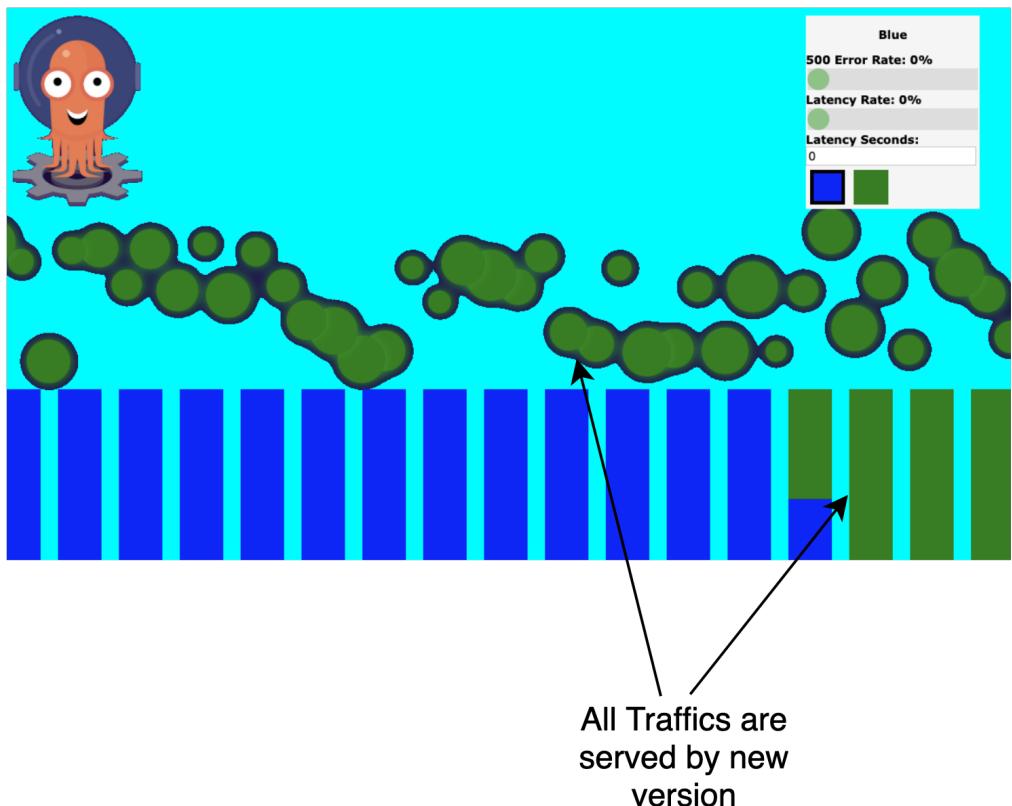


Figure 5.14 After the Green Deployment is complete and Nginx Ingress Controller updated to direct traffic to Green Deployment, the HTML page will start showing both “bubble” and bar chart in green.

If you are happy with the deployment, then you can either delete the Blue Service and Deployment or scale down the Blue Deployment to 0.

Exercise 5.1

How would you scale down the Blue Deployment in a declarative way?

Exercise 5.2

If you want a fast rollback, should you delete the Deployment or scale it down to 0?

5.2.2 Blue-Green with Argo Rollouts

Blue-Green deployment is definitely doable in production using the native Kubernetes Deployment with additional process and automation. A better approach is to make the entire Blue-Green deployment process fully automated and declarative; hence Argo Rollouts is born.

Argo Rollouts introduces a new custom resource called “Rollout” to provide additional deployment strategies such as Blue-Green, Canary (Section 5.3), and Progressive Delivery (Section 5.4) to Kubernetes. The Rollout custom resource provides feature parity with the deployment resource with additional deployment strategies. In the next tutorial, you will see how simple it is to deploy blue-green with Argo Rollouts.

NOTE Please refer to section 5.1.4 on how to enable ingress and install Argo Rollouts in your Kubernetes cluster prior to this tutorial.

1. Deploy the nginx-ingress controller.
2. Deploy the production service and (blue) deployment using Argo Rollouts.
3. Update the manifest to use the green image.
4. Apply the updated manifest to deploy the new version “Green.”

First, we will create the ingress controller, demo-service, and “Blue” deployment.

```
$ kubectl apply -f ingress.yaml
ingress.extensions/demo-ingress created
configmap/nginx-configuration created
$ kubectl apply -f bluegreen_rollout.yaml
rollout.argoproj.io/demo created
service/demo-service created
$ kubectl get ingress
NAME      HOSTS      ADDRESS      PORTS      AGE
demo-ingress  demo.info  192.168.99.111  80        60s
```

Listing 5.8 ingress.yaml.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  rules:
    - host: demo.info
      http:
```

```

paths:
- path: /
  backend:
    serviceName: demo-service
    servicePort: 80
---
apiVersion: v1
data:
  allow-backend-server-header: "true"
  use-forwarded-headers: "true"
kind: ConfigMap
metadata:
  name: nginx-configuration

```

Listing 5.9 bluegreen_rollout.yaml.

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout      #A
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: argoproj/rollouts-demo:blue #B
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
  Strategy:
    blueGreen:      #C
      autoPromotionEnabled: true      #D
      activeService: demo-service   #E
---
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: demo
  type: NodePort

```

#A Specify the kind to be Rollout instead of Deployment

#B Set initial deployment with blue image.
#C Use `blueGreen` instead of Rolling update as deployment strategy.
#D Automatically update the selector in the “demo-service” to send all traffic to green pods.
#E Specify the “demo-service” to front traffic for this rollout object.

NOTE Argo Rollouts internally will maintain one replicaSet for “blue”, one replicaSet for “Green”. It will also ensure that the Green Deployment is fully scaled before updating the service selector to send all traffic over to Green. (Hence only one service is required in this case.) In addition, Argo Rollouts will also wait 30 seconds for all blue traffic to complete and scale down the Blue Deployment.

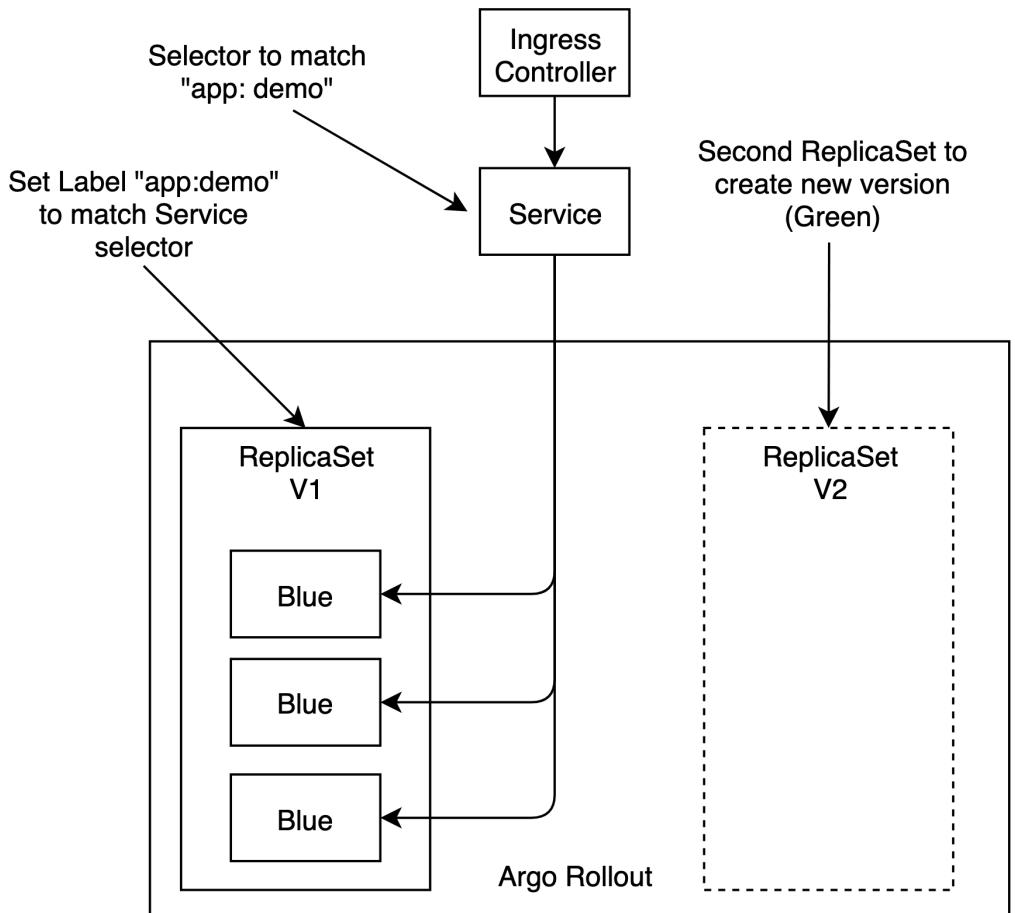


Figure 5.15 Argo Rollout is similar to Deployment which uses one or more ReplicaSet to fulfill deployment requests. In the initial state, Argo Rollout creates ReplicaSet V1 for Blue Pods.

Once you have the ingress controller, service, and deployment created and updated /etc/hosts, you can enter the URL `demo.info` and see the blue service running (Figure 5.13).

NOTE The nginx ingress controller will only intercept traffic with the hostname defined in the custom rule. Please make sure you add “demo.info” and its IP address to your /etc/hosts.

Now we will update the manifest to deploy the new version “Green”. Once you have applied the updated manifest, you can go back to your browser and see all bars and dots turning green (Figure 5.16).

```
$ sed -i .bak 's/demo:blue/demo:green/g' bluegreen_rollout.yaml
$ kubectl apply -f bluegreen_rollout.yaml
deployment.apps/demo configured
service/demo-service unchanged
```

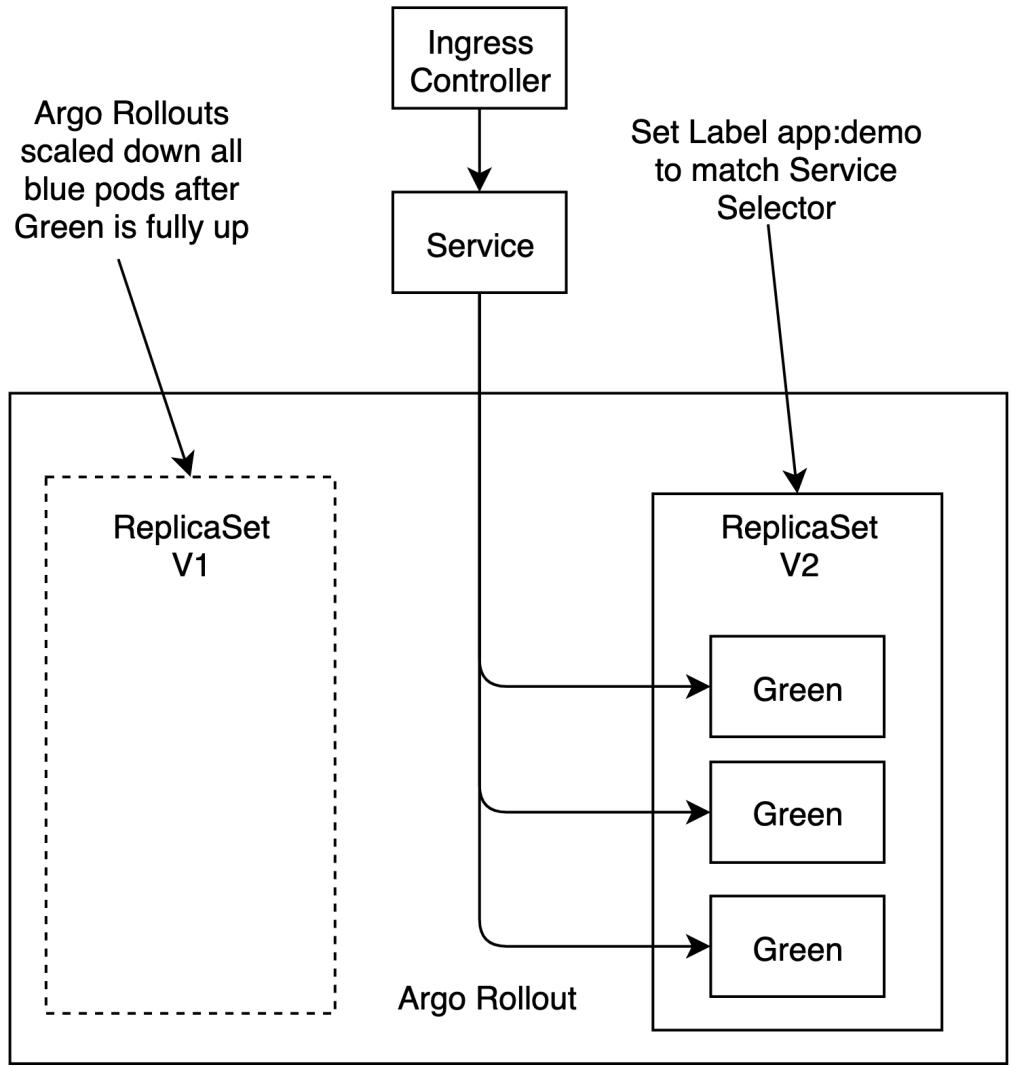


Figure 5.16 Argo Rollout creates ReplicaSet V2 for Green Pods. Once all Green Pods are up and running, Argo Rollout automatically scales down all Blue Pods.

5.3 Canary

Canary deployment is a technique to reduce the risk of introducing a new software version in production by rolling out the change to a small subset of users for a short period before making it available to everybody. Canary acts as an early indicator for failures for avoiding problematic deployments and having a full impact on all customers at once. If one canary

deployment fails, the rest of your servers aren't affected, and you can simply terminate the canary and triage the problems.

NOTE Based on our experience, most production incidents are due to a change in system, such as new deployment. Canary deployment is another opportunity to test your new release before the new release reaches all user populations.

Our Canary example is very similar to the Blue-Green example in 5.2.

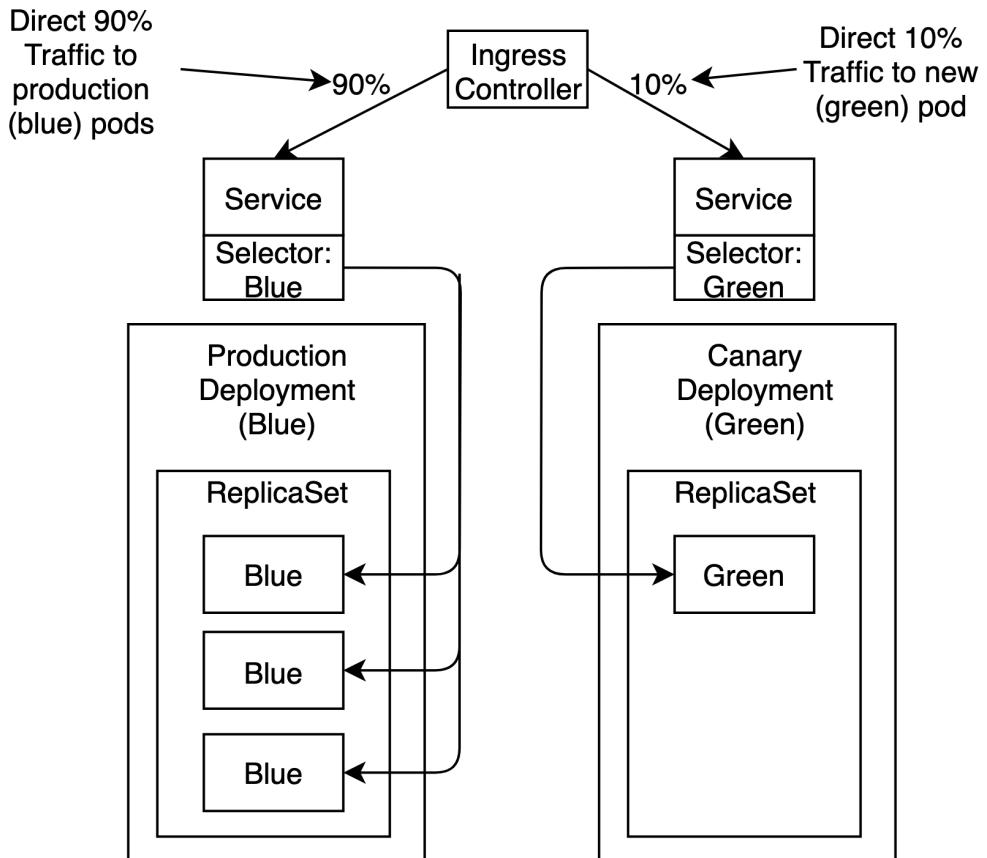


Figure 5.17 Ingress controller will front both blue and green service, but in this case, 90% of traffic will go to the Blue (Production) service, and 10% will go to the Green (Canary) service. Since Green is only getting 10% of the traffic, we will only scale up one Green pod to minimize the resource usage.

With Canary running and getting production traffic, we can then monitor the Canary health (latency, error, etc.) for a fixed period (ex: one hour) to determine whether to scale up

Green Deployment and route all traffic to Green Service or route all traffic back to Blue Service and terminate the Green pod in case of issues.

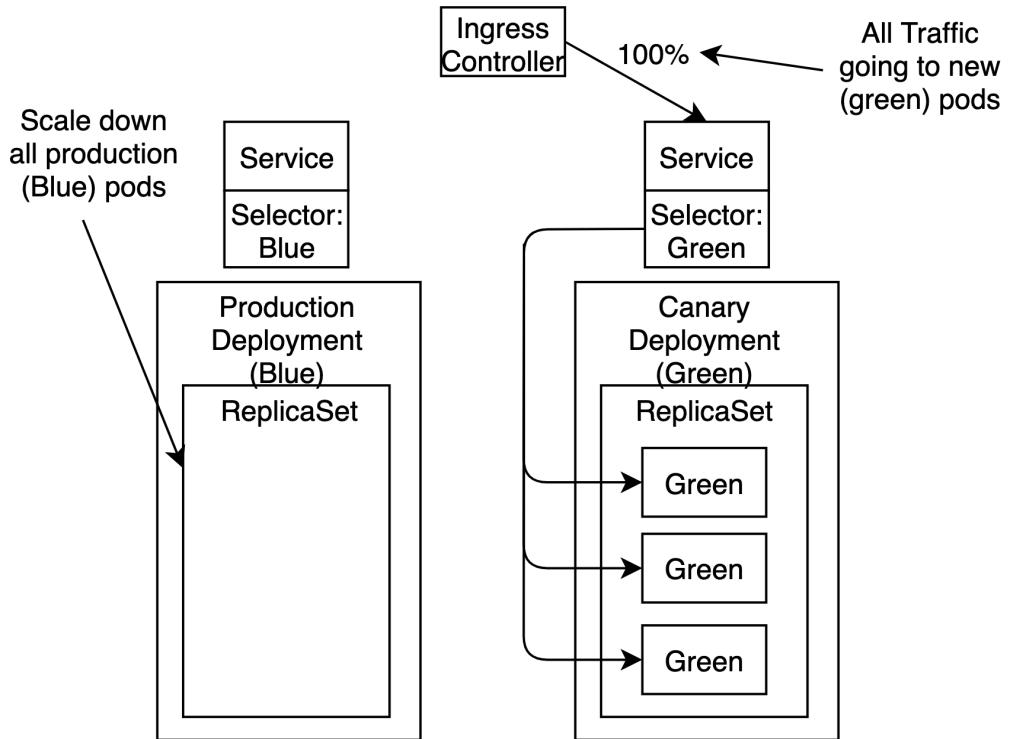


Figure 5.18 If there is no error with the canary pod, Green deployment will scale up to three pods and receive 100% of the production traffic.

5.3.1 Canary with Deployment

In this tutorial, we will perform a Canary Deployment using native Kubernetes Deployment and Service.

NOTE Please refer to section 5.1.4 on how to enable ingress and install Argo Rollouts in your Kubernetes cluster prior to this tutorial.

1. Create the Blue Deployment and Service (Production).
2. Create ingress to direct traffic to blue service.
3. View the application in the browser (blue).
4. Deploy Green Deployment (one pod) and Service and wait for all pods to be ready.
5. Create the canary ingress to direct 10% traffic to green service.
6. View the web page again in the browser (10% green with no error).

7. Scale up the green deployment to 3 pods.
8. Update the canary ingress to send 100% traffic to the green service.
9. Scale down the blue deployment to 0.

We can create the production deployment by applying the `blue_deployment.yaml` (listing 5.4)

```
$ kubectl apply -f blue_deployment.yaml
deployment.apps/blue created
service/blue-service created
```

Now we can expose an ingress controller, so the blue service is accessible from your browser by applying the `blue_ingress.yaml` (listing 5.5). The ‘`kubectl get ingress`’ command will return the ingress controller hostname and IP address.

```
$ kubectl apply -f blue_ingress.yaml
ingress.extensions/demo-ingress created
configmap/nginx-configuration created
$ kubectl get ingress
NAME      HOSTS      ADDRESS      PORTS      AGE
demo-ingress  demo.info  192.168.99.111  80          60s
```

NOTE The nginx ingress controller will only intercept traffic with the hostname defined in the custom rule. Please make sure you add “`demo.info`” and its IP address to your `/etc/hosts`.

Once you have the ingress controller, blue service, and deployment created and updated `/etc/hosts` with “`demo.info`” and correct IP address, you can enter the URL `demo.info` and see the blue service running (Figure 5.9).

Now we are ready to deploy the new Green version. Let’s apply the `green_deployment.yaml` to create the Green service and deployment.

```
$ kubectl apply -f green_deployment.yaml
deployment.apps/green created
service/green-service created
```

Listing 5.10 `green_deployment.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: green
  labels:
    app: green
spec:
  replicas: 1 #A
  selector:
    matchLabels:
      app: green
  template:
    metadata:
      labels:
        app: green
    spec:
      containers:
```

```

    - name: green
      image: argoproj/rollouts-demo:green
      imagePullPolicy: Always
      ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: green-service
  labels:
    app: green
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: green
  type: NodePort

```

#A ReplicaSet is set to 1 for the initial green deployment

Next, we will create the canary_ ingress, so 10% of the traffic is routed to the canary (Green) service.

```
$ kubectl apply -f canary_ingress.yaml
ingress.extensions/canary-ingress configured
configmap/nginx-configuration unchanged
```

Listing 5.11 canary_ingress.yaml.

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"      #A
    nginx.ingress.kubernetes.io/canary-weight: "10"   #B
spec:
  rules:
    - host: demo.info
      http:
        paths:
          - path: /
            backend:
              serviceName: green-service
              servicePort: 80
---
apiVersion: v1
data:
  allow-backend-server-header: "true"
  use-forwarded-headers: "true"
kind: ConfigMap
metadata:
  name: nginx-configuration

```

#A Tells Nginx Ingress to mark this one as “Canary” and associate this ingress with the main ingress by matching host and path.

```
#B Route 10% traffic to green-service
```

Now you can go back to the browser and monitor the green service.

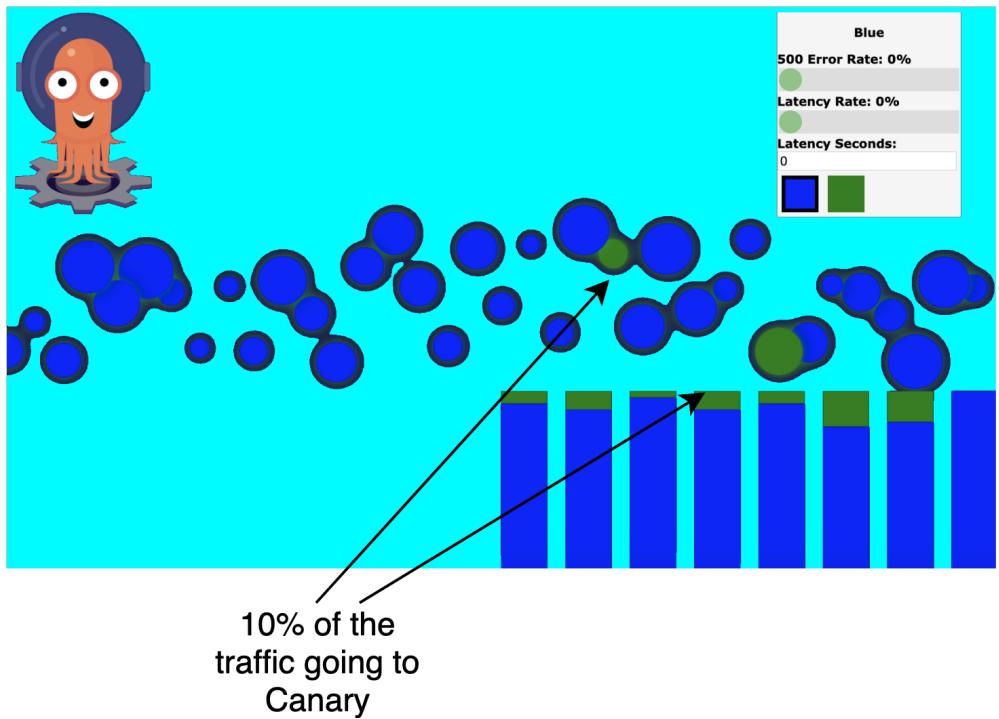


Figure 5.19 The HTML page will show a mix of blue and green in both “bubble” and bar chart because 10% traffic is going to green pods.

If you are able to see the correct result (healthy canary), then you are ready to complete the Canary Deployment (Green Service). We will then scale up the green deployment, send all traffic green service, and scale down the blue deployment.

```
$ sed -i .bak 's/replicas: 1/replicas: 3/g' green_deployment.yaml
$ kubectl apply -f green_deployment.yaml
deployment.apps/green configured
service/green-service unchanged
$ sed -i .bak 's/10/100/g' canary_ingress.yaml
$ kubectl apply -f canary_ingress.yaml
ingress.extensions/canary-ingress configured
configmap/nginx-configuration unchanged
$ sed -i .bak 's/replicas: 3/replicas: 0/g' blue_deployment.yaml
$ kubectl apply -f blue_deployment.yaml
deployment.apps/blue configured
service/blue-service unchanged
```

Now you should be able to see all green bars and dots as 100% of the traffic is routed to the Green Service (Figure 5.20).

NOTE In true production, we will need to ensure all green pods are up before we can send 100% traffic to the canary service. Optionally, we can incrementally increase the percentage of traffic to green service while the green deployment is scaling up.

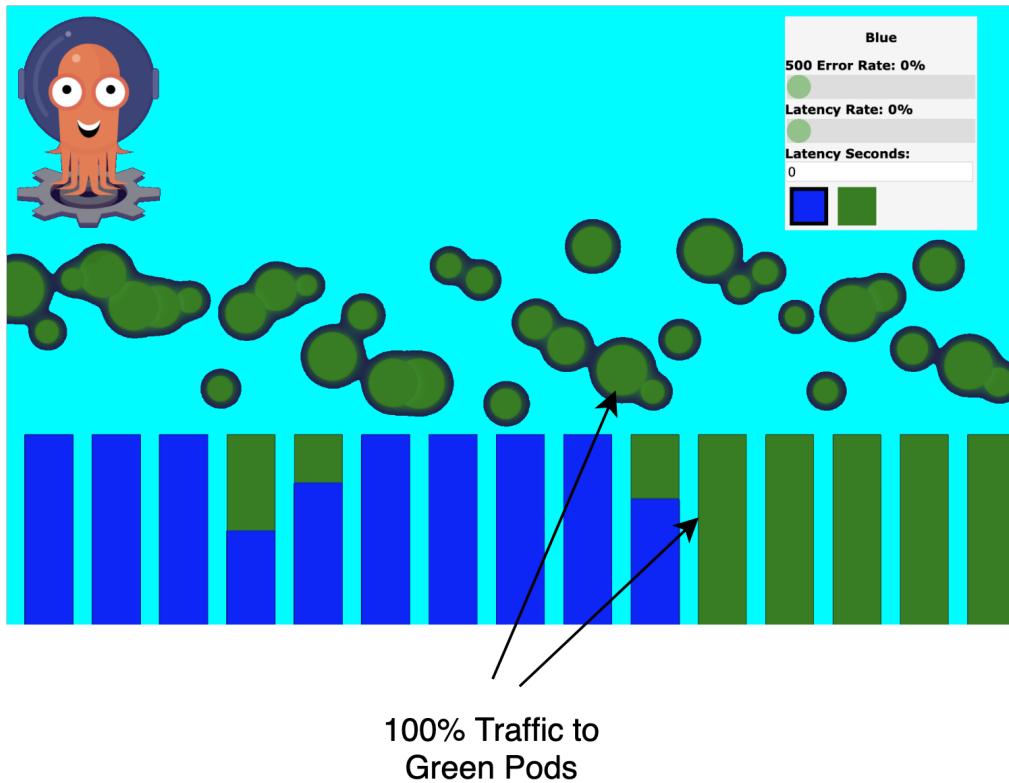


Figure 5.20 If there is no error with the canary, Green deployment will scale up and Blue deployment will scale down. After Blue deployment fully scales down, both “bubble” and bar chart will show 100% green.

5.3.2 Canary with Argo Rollouts

As you can see in 5.3.1, using a Canary Deployment can help detect issues early to prevent problematic deployment but will involve many additional steps in the deployment process. In the next tutorial, we will use Argo Rollouts to simplify the process of Canary deployment.

NOTE Please refer to section 5.1.4 on how to enable ingress and install Argo Rollouts in your Kubernetes cluster prior to this tutorial.

1. Create the ingress, Production Deployment, and Service (Blue).
2. View the application in the browser (Blue).
3. Apply the manifest with Green image with 10% canary traffic for 60 seconds
4. Create the canary ingress to direct 10% traffic to green service.
5. View the web page again in the browser (10% green with no error).
6. Wait 60 seconds
7. View the application again in browser (All Green)

First, we will create the ingress controller (Listing 5.8), demo-service, and “Blue” deployment (Listing 5.12).

```
$ kubectl apply -f ingress.yaml
ingress.extensions/demo-ingress created
configmap/nginx-configuration created
$ kubectl apply -f canary_rollout.yaml
rollout.argoproj.io/demo created
service/demo-service created
$ kubectl get ingress
NAME      HOSTS      ADDRESS      PORTS      AGE
demo-ingress  demo.info  192.168.99.111  80        60s
```

Listing 5.12 canary_rollout.yaml.

```
aapiVersion: argoproj.io/v1alpha1
kind: Rollout
#A
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: argoproj/rollouts-demo:blue
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
  strategy:
    canary:
      maxSurge: "25%" #B
      maxUnavailable: 0
      steps:
        - setWeight: 10
        - pause:
```

```

        duration: 60 #D
---
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: demo
  type: NodePort

```

#A When a Rollout is first deployed, the strategy is ignored and a regular Deployment is performed.

#B Deploy with strategy Canary.

#C Scale-out enough pods to service 10% of the traffic. In this example, Rollout will scale up one Green pod along with the three Blue pods resulting in the Green pod getting 25% of the traffic. Argo Rollout can work with Service Mesh or Nginx Ingress for fine-grain traffic routing.

#D Wait 60 seconds. If no error or user interruption, scale up Green to 100%.

NOTE For the initial deployment (Blue), Rollout will ignore the Canary setting and perform a regular Deployment.

Once you have the ingress controller, service, and deployment created and updated /etc/hosts with the “demo.info” and the correct IP address, you can enter the URL `demo.info` and see the blue service running (Figure 5.9).

NOTE The nginx ingress controller will only intercept traffic with the hostname defined in the custom rule. Please make sure you add “`demo.info`” and its IP address to your /etc/hosts.

Once Blue service is fully up and running, we can now update the manifest with the green image and apply the manifest.

```
$ sed -i .bak 's/demo:blue/demo:green/g' canary_rollout.yaml
$ kubectl apply -f canary_rollout.yaml
rollout.argoproj.io/demo configured
service/demo-service unchanged
```

Once the canary starts, you should see something similar to Figure 5.19 in section 5.3.1. After one minute, Green ReplicaSet will scale up while Blue Deployment scale down with all bars and dots going Green (Figure 5.20 in section 5.3.1).

5.4 Progressive Delivery

Progressive Delivery can also be viewed as a fully automated version of canary deployment. Instead of monitoring for a fixed period (ex: one hour) before scaling up the canary deployment.

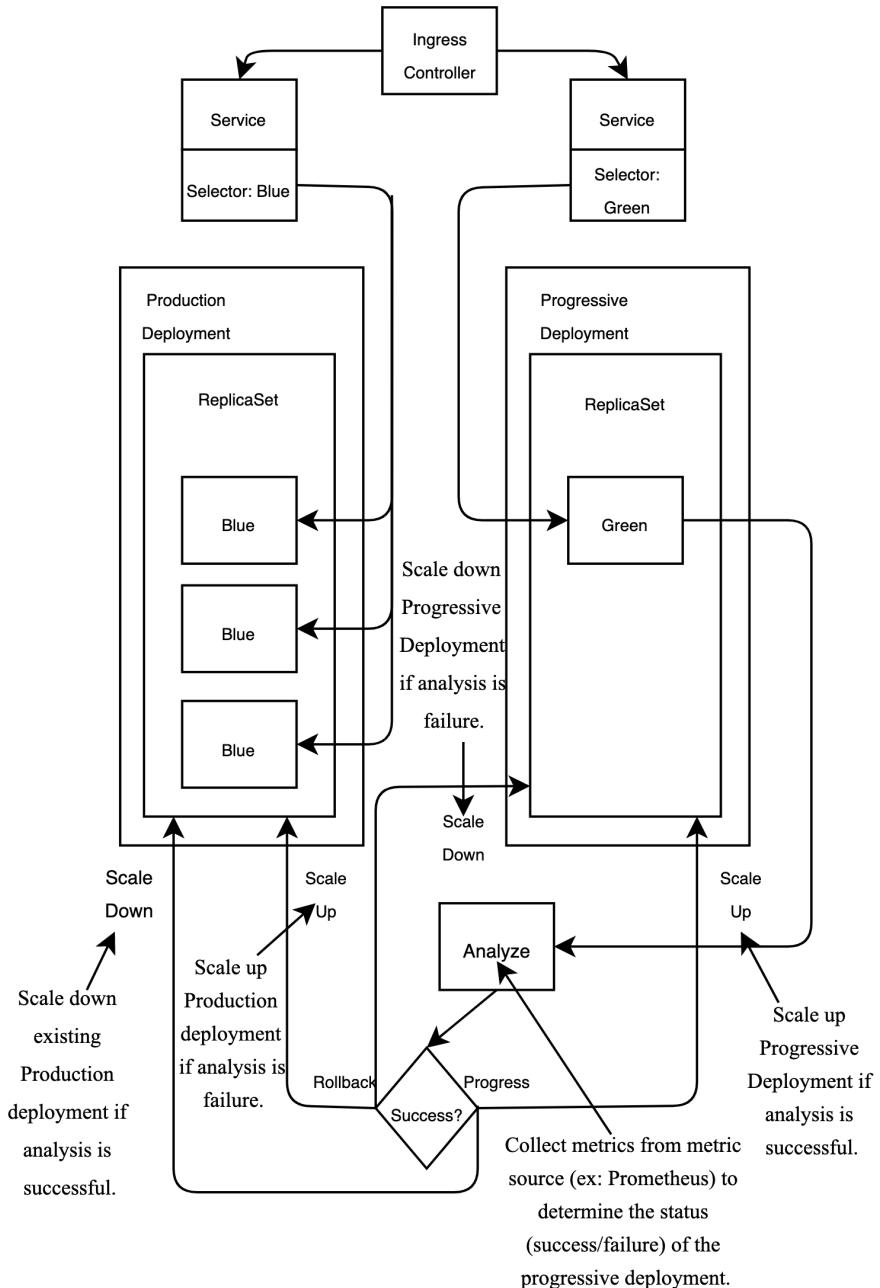


Figure 5.21 Progressive Delivery continuously collects and analyzes the health of the new pods in addition to scale-up the Progressive Deployment (Green) and scale-down the production deployment (blue), as long as the analysis is determined to be successful.

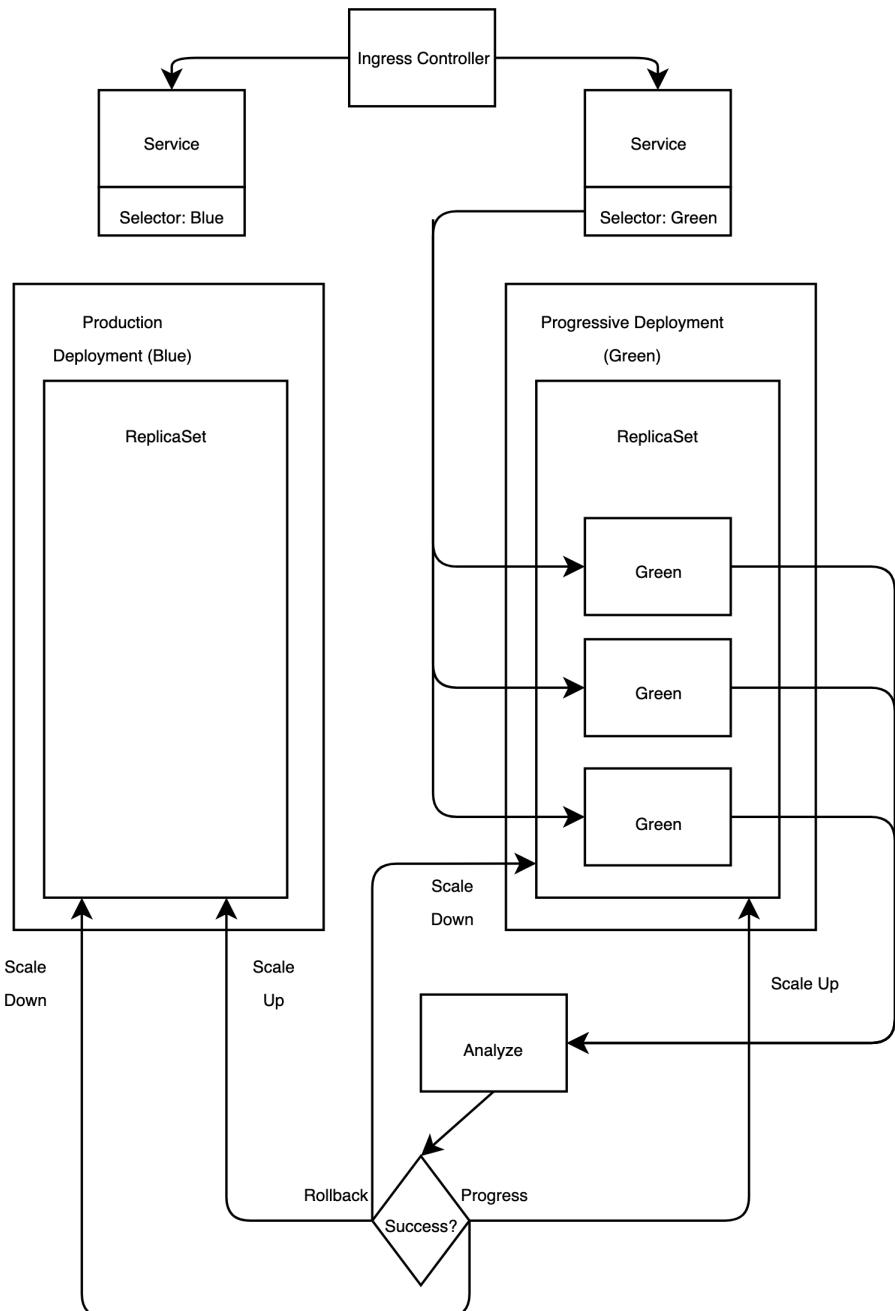


Figure 5.22 depicts a completed Progressive Delivery with fully scaled-up Green Deployment and scaled-down

Blue Deployment.

5.4.1 Progressive Delivery with Argo Rollouts

Kubernetes does not provide any analysis tool to determine the correctness of the new deployment. In this tutorial, we will use Argo Rollouts to implement the Progressive Delivery. Argo Rollouts uses the Canary Strategy along with Analysis Templates to achieve Progressive Delivery.

NOTE Please refer to section 5.1.4 on how to enable ingress and install Argo Rollouts in your Kubernetes cluster prior to this tutorial.

1. Create the Analysis Templates.
2. Create the ingress, Production Deployment, and Service (Blue).
3. Create ingress to direct traffic to production service.
4. View the application in the browser (Blue).
5. Update and apply the manifest with Green image with the “Pass” template.
6. View the web page again in the browser (Green).
7. Update and apply the manifest with Green image with the “Fail” template.
8. View the application again in the browser (Still Blue!)

First, we will create the Analysis Templates (Listing 5.13) on how Rollout can collect metrics to determine the health of the pods. For simplicity, we will create one AnalysisTemplate `pass`, which will always return 0 (healthy) and another AnalysisTemplate `fail`, which will always return 1 (unhealthy). In addition, Argo Rollout internally maintains multiple replicaSet, so there is no need for multiple services. Next, we will create the ingress controller (Listing 5.8), demo-service, and “Blue” deployment (Listing 5.14).

NOTE For production, AnalysisTemplate has support for Prometheus, Wavefront, Netflix Kayenta, or can be extended for other metric stores.

```
$ kubectl apply -f analysis-templates.yaml
analysistemplate.argoproj.io/pass created
analysistemplate.argoproj.io/fail created
$ kubectl apply -f ingress.yaml
ingress.extensions/demo-ingress created
configmap/nginx-configuration created
$ kubectl apply -f rollout-with-analysis.yaml
rollout.argoproj.io/demo created
service/demo-service created
$ kubectl get ingress
NAME          HOSTS      ADDRESS      PORTS   AGE
demo-ingress  demo.info  192.168.99.111  80      60s
```

Listing 5.13 analysis-templates .yaml.

```
apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
```

```

name: pass
spec:
  metrics:
    - name: pass
      interval: 15s          #A
      failureLimit: 1
    provider:
      job:
        spec:
          template:
            spec:
              containers:
                - name: sleep
                  image: alpine:3.8
                  command: [sh, -c]
                  args: [exit 0]      #B
                  restartPolicy: Never
            backoffLimit: 0

---
apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: fail
spec:
  metrics:
    - name: fail
      interval: 15s          #C
      failureLimit: 1
    provider:
      job:
        spec:
          template:
            spec:
              containers:
                - name: sleep
                  image: alpine:3.8
                  command: [sh, -c]
                  args: [exit 1]      #D
                  restartPolicy: Never
            backoffLimit: 0

```

#A Run for 15 seconds
#B Return 0 (Always Pass)
#C Run for 15 seconds
#D Return 1 (Always Fail)

Listing 5.14 rollout-with-analysis.yaml.

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: demo
spec:
  replicas: 3
  revisionHistoryLimit: 1
  selector:
    matchLabels:
      app: demo

```

```

strategy:
  Canary:                      #A
    analysis:
      templateName: pass       #D
      steps:
        - setWeight: 10         #B
        - pause:
          duration: 20          #C
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - image: argoproj/rollouts-demo:blue
          imagePullPolicy: Always
          name: demo
          ports:
            - containerPort: 8080
  ---
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    app: demo
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  selector:
    app: demo
  type: NodePort

```

#A Deploy with strategy Canary.

#B Scale-out enough pods to service 10% of the traffic. In this example, Rollout will scale up one Green pod along with the three Blue pods resulting in the Green pod getting 25% of the traffic. Argo Rollout can work with Service Mesh or Nginx Ingress for fine-grain traffic routing.

#C Wait 20 seconds before full scale-up.

#D Specify the Analysis Template “Pass”.

NOTE For the initial deployment (Blue), Rollout will ignore the Canary setting and perform a regular deployment.

Once you have the ingress controller, service, and deployment created and updated /etc/hosts with the “demo.info” and the correct IP address, you can enter the URL `demo.info` and see the blue service running (Figure 5.9).

Once Blue service is fully up and running, we can now update the manifest with the green image and apply the manifest. You should see the blue “progressive” turning green and completely green after 20 seconds (Figure 5.17).

```
$ sed -i .bak 's/demo:blue/demo:green/g' rollout-with-analysis.yaml
$ kubectl apply -f rollout-with-analysis.yaml
rollout.argoproj.io/demo configured
```

```
service/demo-service unchanged
```



Figure 5.23 As Green Deployment progressively scales up, both “bubble” and bar chart will gradually become all green if there is no error.

Now let's deploy again and back to the blue image. This time we will also switch to the “Fail” analysis template that will return failure status after 15 seconds. We should see the “blue” progressively appearing on the browser but back to green after 15 seconds (Figure 5.18).

```
$ sed -i .bak 's/demo:green/demo:blue/g' rollout-with-analysis.yaml
$ sed -i .bak 's/templateName: pass/templateName: fail/g' rollout-with-analysis.yaml
$ kubectl apply -f rollout-with-analysis.yaml
rollout.argoproj.io/demo configured
service/demo-service unchanged
```

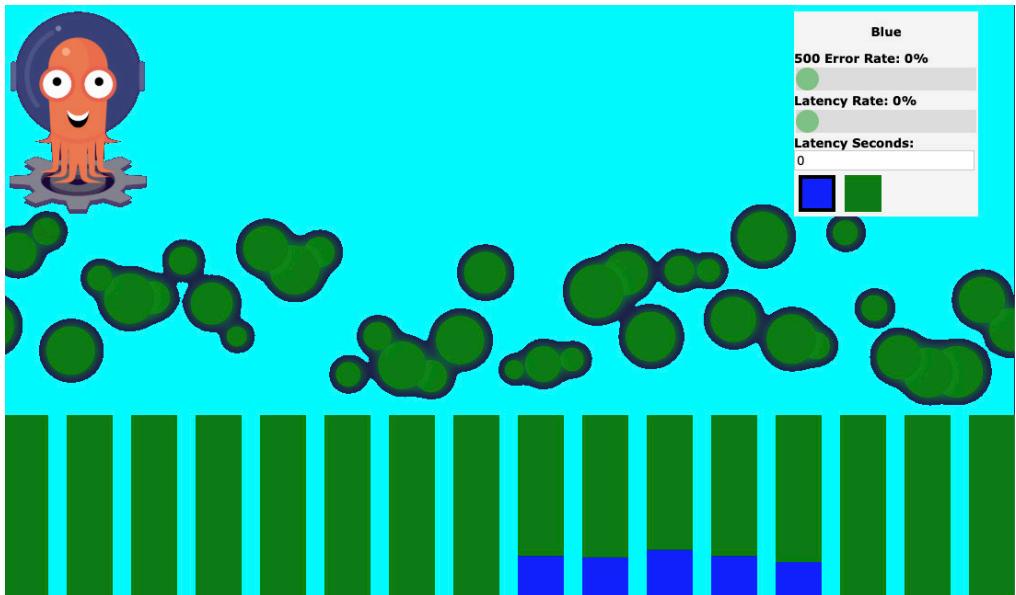


Figure 5.24 As Blue Deployment progressively scales up but returning errors, the Blue Deployment will get scaled down during failure and both “bubble” and bar chart return to Green.

NOTE The “demo” Rollout will be marked as “aborted” after the failure and will not deploy again until the abort state is set to false.

From this tutorial, you can see that Argo Rollout uses the Canary Strategy to “progressively” scale up the Green Deployment if the Analysis Template continues to report success based on the metrics collected. If Analysis Template reports failures, Argo Rollout will “rollback” by scaling down the Green Deployment and scaling up Blue Deployment back to its original state.

Table 5.1 Deployment Strategy Comparison

	Pros	Cons
Deployment	Build in with Kubernetes. Rolling Update. Minimal hardware required.	Backward compatible and stateless application only.
Blue-Green	Work with stateful application and non-backward compatible deployment.	Require additional automation Require twice the hardware during the

	Fast rollback.	deployment.
Canary	Verify new releases using production traffic and dependencies with a subset of users.	Require additional automation. Longer deployment process. Backward compatible and stateless application only.
Progressive	Gradually deploy new releases to a subset of users and continuously scale out to all users if metrics are good. Automatic rollback if metrics are bad.	Require additional automation. Require metrics collection and analysis. Longer deployment process. Backward compatible and stateless application only.

5.5 Summary

- ReplicaSet is not declarative and not suited for GitOps.
- Deployment is fully declarative and complementary for GitOps.
- Deployment performs Rolling update and is best for stateless and backward compatible deployment.
- Deployment can be customized with Max Surge to specify the number of new pods and Max Unavailable to limit the number of pods being terminated.
- Blue-Green Deployment is suitable for non-backward deployment or sticky session service.
- Blue-Green deployment can be implemented natively by leveraging two Deployments (each with custom label) and updating selector in Service to route 100% of the traffic to the active deployment.
- Canary acts as an early indicator for failures for avoiding problematic deployments and having a full impact on all customers at once.
- Canary deployment can be implemented natively by leveraging two Deployments and a nginx ingress controller to “dial” the traffic gradually.
- Progressive Delivery is an advanced version of Canary using real-time metrics to continue or abort the deployment.
- Argo Rollouts is an open source project which can simplify Blue-Green, Canary and Progressive deployment.
- Each deployment strategy has its pros/cons (Table 5.1) and it is important to select the “right” strategy for your application.

6

Access Control & Security

This chapter covers:

- The areas of attack when using GitOps driven deployment
- Ensuring critical infrastructure components are protected
- Guidelines for choosing the right configuration management pattern
- Enhancing security to avoid security pitfalls in GitOps

Access control and security topics are always essential and are especially crucial for deployment and infrastructure management. In this case, the attack surface includes expensive things like infrastructure, dangerous things like policy and compliance, and the most important things like data stores that contain user data. Modern operations methodologies enable engineering teams to move at a much quicker pace and optimize for fast iterations. However, more releases also mean more chances of introducing vulnerabilities and leads to new challenges for security teams. Traditional security processes that rely on human operational knowledge may still work but struggle to scale and meet the needs of enterprises utilizing GitOps with automated build and release infrastructure.

We recommend you read chapters 1 and 2 before reading this chapter.

6.1 Introduction to Access Control

The security topic is both critical and complex. Usually, it is handled by a security specialist or even a whole, dedicated security team. So why do we talk about it while discussing GitOps? The GitOps changes security responsibilities in the same way it has altered operational responsibility boundaries. With GitOps and Kubernetes, the engineering team is empowered to contribute to the security by writing Kubernetes access configs and using Git to enforce proper configuration change processes. Given that the security team is no longer a bottleneck, it can offload some responsibilities to developers and focus on providing security infrastructure. GitOps facilitates a tighter and more productive collaboration between security

engineers and DevOps engineers, allowing any proposed changes that impact an environment's security to undergo proper security reviews and approval before it affects production.

6.1.1 What is Access Control?

To better understand the nuances of access control in combination with GitOps, let's learn what access control is first.

Access control is a way of limiting access to a system or to physical or virtual resources. It dictates who is allowed to access the protected resources and what operations are allowed to be performed. Access control is composed of two parts; authentication, ensuring that users are who they say they are, and authorization, ensuring they have appropriate access to perform the requested action against the specified resources.

Regardless of the domain area, access control includes three main components: subject, object, and reference monitor.

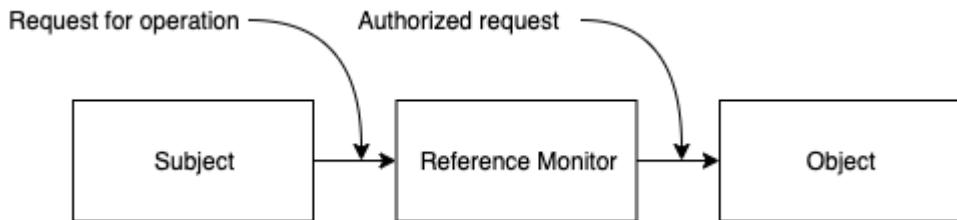


Figure 6.1 The subject is the entity that requests access to an object. The object is the entity or resource being accessed. The reference monitor is the entity controlling access to the protected object.

The most straightforward demonstration of the access control system is a physical world example: a person trying to enter the building through the door. The person is a subject who is trying to access the object, which is a building. The door is a reference monitor that will authorize the access request only if the person trying to enter has the door key.

Exercise 6.1

An email client is trying to read emails from an email server. Can you identify what the subject, object, and reference monitor are for this scenario?

6.1.2 What to Secure?

When securing the application delivery process to a Kubernetes cluster end-to-end, many different components need to be secured. These include (but are not limited to):

1. CI/CD Pipeline
2. Container Registry
3. Git Repository
4. Kubernetes Cluster
5. Cloud Provider or Data Center

6. The application itself
7. GitOps Operator (if applicable)

Each of these components has its unique security concerns, authentication mechanisms, RBAC (Role-Based Access Control) models, and will be configured differently depending on many factors and considerations. Since security is only as strong as the weakest link, all components play an equally important role in the cluster's overall security.

Generally speaking, security is often a balancing act between security and convenience. A system that might be extremely secure may be so inconvenient that it becomes unusable from a user perspective. As an operator, the goal is to make the user experience as convenient as reasonably possible without compromising security.

Some considerations which factor into the security of the components include:

1. What are the potential attack vectors?
2. What are the worst-case consequences if the component becomes compromised?
3. Who should be allowed access to the service?
4. What permissions (RBAC) will various individuals users have when allowed access?
5. What protections can be put in place to mitigate risk?

The next few sections will describe each component and some of the unique security considerations.

CI/CD PIPELINE

The CI/CD build and deployment pipeline is the starting point for delivering freshly built software to the Kubernetes cluster. Jenkins, Circle CI, and Travis CI are examples of some popular CI/CD systems. It's frequently an afterthought of things to secure since most thought and energy is focused on protecting the production environment and production data. Still, the CI/CD is an equally important piece of the puzzle. This is because the CI/CD pipeline ultimately controls *how* new software is driven to the environment. When compromised, it has the capability of delivering *harmful* software to the cluster.

A build system is generally configured with sufficient credentials to perform its duties. For example, to publish new container images, the CI/CD pipeline might require credentials to the container registry. Traditionally, the build system is also given access and credentials to the Kubernetes cluster to perform the actual deployment. But as we will see later in this chapter, direct access to the cluster is no longer necessary with the advent of GitOps.

An attacker with access to the CI/CD build system could compromise security in many ways. For example, a pipeline could be modified to expose the container registry or cluster credentials mentioned above. Another example is the pipeline could become hijacked such that it deploys malicious containers into the cluster instead of the intended one.

There are even some scenarios in which a bad actor might compromise security by merely using the CI system's standard functionality. For example, when a pull request is made against a code repo, it kicks off a pipeline performing a series of steps to validate and test the change. The contents of these steps are typically defined in a file contained in the code repo (e.g., a Jenkins' **Jenkinsfile** or Circle CI's **.circleci/config.yml**). The ability to open new pull requests is often open to the world so that anyone can propose contributions to the project. However, an attacker could simply open a pull request that modifies the

pipeline to do something malicious. For this reason, many CI systems incorporate features to prevent pipelines from being executed when an untrusted source makes the PR.

CONTAINER REGISTRY

A container registry houses the container images that will be deployed in the cluster. Because the container images in the registry have the potential of running in clusters, the contents of the registry need to be trusted, as well as the users who can push to that registry. Because anyone can publish images to public registries such as DockerHub, Quay.io, and grc.io, it is a standard security measure in enterprises to block pulling images from these untrusted container registries entirely. Instead, all images would be pulled from an internal, trusted registry, which could be periodically scanned for vulnerabilities in the repositories.

An attacker with privileges to the trusted container registry could push images to the registry and overwrite existing, previously trusted images. For example, say your cluster is already running some image **mycompany/guestbook:v1.0**. An attacker with access to the registry could push a new image and overwrite the existing **guestbook:v1.0** tag, changing the meaning of that image to something malicious. Then, when the next time the container starts (e.g., due to pod rescheduling), it would run a compromised version of the image.

This attack might go undetected since, from the perspective of Kubernetes and the GitOps system, everything is as expected -- the live manifests match the configuration manifests in git. To combat this issue, image tags (or image versions) can be designated as "immutable" in some registries so that once written, the meaning of that image tag can never change.

NOTE *Immutable Image Tags* Some image registries (such as DockerHub) provide a feature which makes image tags immutable. This means that once an image tag already exists, no one can overwrite it, essentially preventing image tags from being re-used. The use of this feature adds additional security by preventing existing deployed image tags from being modified.

GIT REPOSITORY

The Git repository, in the context of GitOps, defines *what* resources will be installed into the cluster. The Kubernetes manifests, which are stored in the git repo, are the ones that ultimately end up in the cluster. Therefore, anyone who can access the Git repository should be trusted in deciding the cluster's makeup, including things like Deployments, container images, Roles, RoleBindings, Ingresses, and NetworkPolicies.

In the worst case, an attacker with full access to the Git repository could push a new commit to the Git repo updating a Deployment to run a malicious container in the cluster. They might also add a Role and RoleBinding that could grant the Deployment enough privileges to read Secrets and exfiltrate sensitive information.

The good news is that since an attacker would need to push commits to the repository, the malicious actions performed would be done in plain sight and could be audited and traced. However, commit and Pull Request merge access to the Git repository should be restricted to a limited set of people which would effectively have full cluster administration access.

KUBERNETES CLUSTER

Securing a Kubernetes Cluster is deserving of a book onto itself, so we will only aim to cover the topics most relevant to GitOps. As you are aware, the Kubernetes cluster is the infrastructure platform that runs your application code. An attacker who has gained access to the cluster is arguably the worst-case scenario. For this reason, Kubernetes clusters are extremely high-value targets for attackers, and the security of the cluster is paramount.

GitOps enables a whole new set of options on how you might decide to grant users access to the cluster. This will be covered in depth later in the chapter but, at a high level, GitOps gives operators a new way of providing access to the cluster (i.e., through git), as opposed to the traditional method of giving users direct access to the cluster (e.g., with a personalized kubeconfig file).

Traditionally, before GitOps, developers would generally require direct access to the Kubernetes cluster to manage and make changes to their environment. But with GitOps, direct access to the cluster is no longer strictly required since environment management can go through a new alternate medium, git. And suppose all developer access to the cluster can be via Git. In that case, it also means that operators can decide to close traditional, direct access to the cluster completely (or at least write access) and enforce all changes to go through Git.

CLOUD PROVIDER OR DATA CENTER

Perhaps out-of-scope in context GitOps, but nonetheless important to the discussion around security is the actual underlying cloud provider (e.g., AWS) or physical datacenter in which the Kubernetes clusters are running in. Commonly, an application that runs in Kubernetes will depend on some number of managed resources or services in the cloud for things like databases, DNS, object storage (e.g., S3), message queues, and many others. Because both developers and applications need access to these resources, an operator needs to consider how creation and access to these cloud provider resources may be granted to these users.

A developer will likely require access to their database to perform things like database schema migrations or generating reports. While GitOps in itself does not provide a solution for securing the database per se, GitOps does come into play when the database configuration invariably starts creeping into the Kubernetes manifests (which are managed via GitOps). For example, one mechanism an operator might employ to help secure access to a database is IP whitelisting in a Kubernetes NetworkPolicy. And since NetworkPolicy is just a standard Kubernetes resource that can be managed via git, the *contents* (IP whitelist) of the NetworkPolicy becomes significant to operators as a security concern.

A second consideration is that Kubernetes resources can have a profound impact on cloud provider resources. For example, a user who is allowed to create ordinary Kubernetes Service objects could cause many costly load balancers to be created in the cloud provider and unintentionally exposing services to the outside world. For these reasons, it's imperative that cluster operators have a deep understanding of the relationship between Kubernetes resources and cloud provider resources and the consequences of allowing users to manage these resources on their own.

GITOPS OPERATOR

Depending on your choice of a GitOps operator, securing the operator may or may not be an option. An elementary GitOps operator, such as our “poor-mans” CronJob-based GitOps operator example from chapter 2, has no other security implications since it is not a service that can be exposed externally, nor does it have any form of management aspect to it. On the other hand, a tool such as Argo CD, Helm, or Jenkins X is intended to be exposed to end-users. As a result, it has additional security considerations since it could be a vector of attack.

6.1.3 Access control in GitOps

First of all, let’s figure out the access control subjects and objects in the continuous deployment security model. As we’ve learned already, the objects are resources that have to be protected. The continuous deployment surface attack is large, but the idea of immutable infrastructure and Kubernetes narrow it to just two things: Kubernetes configuration and deployment artifacts.

As you already know, the Kubernetes configuration is represented by a collection of Kubernetes resources. The resource manifests are stored in Git and automatically applied to the target Kubernetes cluster. The deployment artifacts are container images. Having these two, you can shape your production in any way and even recreate it from scratch at any time.

The access control subjects, in this case, are engineers and automated processes, such as CI, pipeline. The engineers are leveraging automation to continuously produce new container images and update Kubernetes configuration to deploy them.

Unless you are using GitOps, the Kubernetes configuration is updated either manually or scripted in continuous integration. This approach, sometimes called CI Ops⁴¹, usually makes the security team very nervous.

⁴¹ <https://www.weave.works/blog/kubernetes-anti-patterns-let-s-do-gitops-not-ciops>

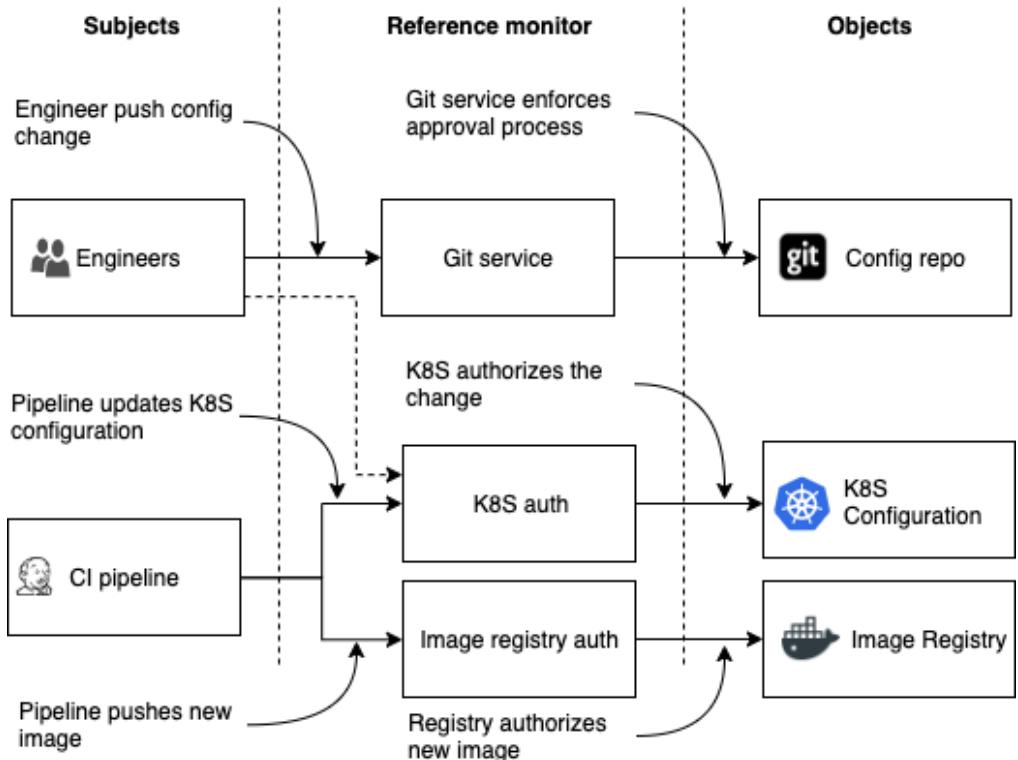


Figure 6.2 CI Ops security model is not safe since it provides cluster access to both engineers and continuous integration systems. The problem here is that the CI system gets control over the cluster and is allowed to make arbitrary Kubernetes configuration changes. That significantly expands the attack surface and makes it difficult to secure your cluster.

So how does GitOps improve the situation? GitOps unifies the process of applying changes from Git repo to the cluster. That allows keeping access tokens closer to the cluster and effectively moves the burden of securing access to the Git repository.

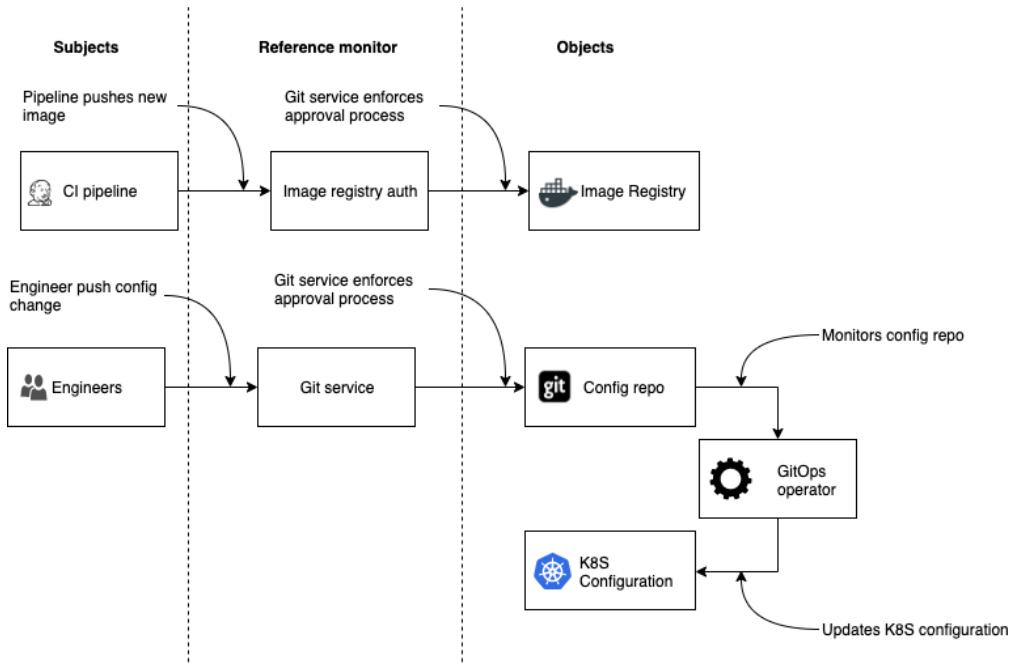


Figure 6.3 GitOps security model limits cluster access to the GitOps operator only. Attack surface is greatly reduced and protecting the cluster is much simpler.

Protecting configuration in the Git repository still requires an effort. The great thing about it is that we can use the same tools that are used to protect the application source code. Git hosting providers such as Github and Gitlab allow us to define and enforce the change process like mandatory reviews or static analysis for every change. Since the GitOps operator is the only subject with cluster access, it is much easier to define what can and cannot be deployed into the cluster by the engineering team and significantly improves cluster security.

Let's go ahead and learn what it takes to protect Kubernetes configs in the Git repository and how to fine-tune Kubernetes access control.

6.2 Access Limitations

As discussed at the beginning of the chapter, there are many components involved in securing the cluster, including the CI/CD build system, the container registry, and the actual Kubernetes cluster. Each component implements specific access control mechanisms to allow or deny access.

6.2.1 Git repository access

Git is an extremely developer-oriented tool. By default, it is configured to make it extremely easy to change anything at any time. This simplicity is what made Git so popular in the developer's community.

However, Git is built on top of the solid cryptographic foundation: it uses Merkle trees as a fundamental underlying data structure. The same data structure is used as a foundation for the blockchain⁴². As a result, Git can be used as a distributed ledger, making it a great audit log storage.

NOTE Merkle Tree⁴³ Hash tree or Merkle tree is a tree in which every leaf node is labeled with the hash of a data block, and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes.

Here is a brief overview of how Git works. Every time the developer wants to save a set of changes, Git calculates an introduced diff and creates a bundle that includes introduced changes, various metadata fields, such as the date and author, as well as a reference to the parent bundle that represents the previous repository state. Finally, the bundle is hashed and stored in a repository. That bundle is called a commit. That algorithm is pretty much the same algorithm that is used in the blockchain.

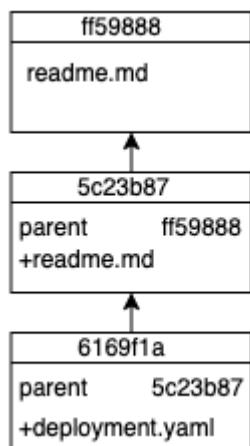


Figure 6.4 Each Git commit references the previous commit and forms a tree-like data structure. All modifications are fully tracked in the Git repository.

The hash is used as a guarantee that the code used is the same code that was committed, and it has not been tampered with. So the Git repository is a chain of commits which is cryptographically protected from hidden modifications. It is safe to trust the Git

⁴² <https://en.wikipedia.org/wiki/Blockchain>

⁴³ https://en.wikipedia.org/wiki/Merkle_tree

implementation thanks to cryptographic algorithms behind it, so we can use it as the audit log.

CREATE DEPLOYMENT REPOSITORY

Let's create a sample deployment repository and then see what it takes to make it ready to drive GitOps deployment. For your convenience, let's use the existing deployment repository available at <https://github.com/gitopsbook/sample-app-deployment>. The repository contains the deployment manifests of a Kubernetes Service and Deployment resource. The Deployment resource manifest is available in the following listing:

Listing 6.1 Sample App Deployment (`deployment.yaml`).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - image: gitopsbook/sample-app:v0.1
          name: sample-app
          command:
            - /app/sample-app
          ports:
            - containerPort: 8080
```

As has been mentioned, Git is a distributed version control system. That means every developer has a full local repository copy with full access to make changes. However, there is also a common repository that all team members use to exchange their changes. That common remote repository is hosted by the Git hosting service like GitHub or GitLab. Hosting services provide a set of security features that allow protecting repository from unwanted modifications, enforcing commit author identity, preventing history override, and many more.

As a first step, please navigate to the **gitopsbook/sample-app-deployment** repo and create a fork⁴⁴ in your Github account.

```
https://github.com/gitopsbook/sample-app-deployment
```

Once the fork is created, use the following command to clone the repository locally and get ready to make changes:

```
$ git clone https://github.com/<username>/sample-app-deployment.git
```

⁴⁴ <https://help.github.com/en/github/getting-started-with-github/fork-a-repo>

```
Cloning into 'sample-app-deployment'...
remote: Enumerating objects: 14, done.
remote: Total 14 (delta 0), reused 0 (delta 0), pack-reused 14
Receiving objects: 100% (14/14), done.
Resolving deltas: 100% (3/3), done.
```

Although the repository is public, that does not mean that everyone with the Github account can make a change without proper permission. Github ensures that a user is either a repository owner or invited as a collaborator⁴⁵.

Exercise 6.1

Clone the repository using the HTTPS URL and try to push any changes without providing your Github username and password.

```
git clone https://github.com/<username>/sample-app-deployment.git
```

Instead of creating a personal repository, you might create an organization⁴⁶ and manage access using teams. That set of access management features is very comprehensive and covers most of the use-cases, starting from the single developer to the large organization. However, this is not enough.

Exercise 6.2

Create a second Github user and invite that user as a collaborator. Try pushing any changes using the second Github user credentials.

ENFORCE A CODE REVIEW PROCESS

Neither the cryptographic protection nor authorization setting can protect from vulnerabilities introduced intentionally by a malicious developer or simply by mistake through poor coding practices. Whether the vulnerabilities in application source code are introduced intentionally or not, the recommended solution is the same; all changes in the deployment repository must go through a code review process that is enforced by the Git hosting provider.

Let's make sure that every change to the **master** branch of the sample-app-deployment repository goes through the code review process and approved by at least one reviewer. The steps to enable mandatory review process are described at <https://help.github.com/en/github/administering-a-repository/enabling-required-reviews-for-pull-requests> :

- Navigate to the 'Branches` section in the repository settings.
- Click the `Add Rule` button.
- Enter the required branch name.
- Enable 'Require pull request reviews before merging` and `Include administrators` settings.

Next, let's try to make a config change and push it to the master:

```
$ sed -i .bak 's/v0.1/v0.2/' deployment.yaml
$ git commit -am 'Upgrade image version'
```

⁴⁵ <https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/inviting-collaborators-to-a-personal-repository>

⁴⁶ <https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/managing-your-membership-in-organizations>

```
$ git push
remote: error: GH006: Protected branch update failed for refs/heads/master.
remote: error: At least 1 approving review is required by reviewers with write access.
To github.com:<username>/sample-app-deployment.git
 ! [remote rejected] master -> master (protected branch hook declined)
error: failed to push some refs to 'github.com:<username>/sample-app-deployment.git'
```

The git push fails with the error message required to create a pull request and get an approval. This guarantees that at least one additional person is going to review the change and sign off the deployment.

Don't forget to run clean up before moving to the next paragraph. Please delete the rule that protects the master branch in GitHub and run the following command to reset local changes:

```
$ git reset HEAD^1 --hard
```

Exercise 6.3

Explore additional settings under the `Require pull request reviews before merging` section. Think which settings combination is suitable for your project or organization.

ENFORCE AUTOMATED CHECKS

In addition to human judgment, Pull Requests allows us to incorporate an automated manifests analysis that can help to catch security issues very early. Although the ecosystem of Kubernetes security tools is still emerging, there are already several options available. A good example is Kubeaudit⁴⁷ and Kubesec⁴⁸. Both tools are available under the Apache license and allow scanning Kubernetes manifests to find weak security parameters.

Because our repository is open-source and hosted by Github, we can use a continuous integration service such as <https://travis-ci.org/> or <https://circleci.com/> for free! Let's configure an automated Kubeaudit usage and enforce successful verification for every Pull Request using <https://travis-ci.org/>.

```
git add .travis.yml
git commit -am 'Add travis config'
git push
```

Listing 6.2 .travis.yml.

```
language: bash
install:
  - curl -sLf -o kubeaudit.tar.gz
    https://github.com/Shopify/kubeaudit/releases/download/v0.7.0/kubeaudit_0.7.0_linux_amd64.tar.gz
  - tar -zxf kubeaudit.tar.gz
  - chmod +x kubeaudit
script:
  - ./kubeaudit nonroot -f deployment.yaml &> errors
  - if [ -s errors ] ; then cat errors; exit -1; fi
```

⁴⁷ <https://github.com/Shopify/kubeaudit>

⁴⁸ <https://kubesec.io/>

Once configuration is ready we just need to enable CI integration at <https://travis-ci.org/<username>/sample-app-deployment> and create a pull request:

```
$ git checkout -b change1
Switched to a new branch 'change1'

$ sed -i .bak 's/v0.1/v0.2/' deployment.yaml

$ git commit -am 'Upgrade image version'
[change1 c52535a] Upgrade image version
 1 file changed, 1 insertion(+), 1 deletion(-)

$ git push --set-upstream origin change1
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 359 bytes | 359.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'change1' on GitHub by visiting:
remote:     https://github.com/<username>/sample-app-deployment/pull/new/change1
remote:
To github.com:<username>/sample-app-deployment.git
 * [new branch]      change1 -> change1
Branch 'change1' set up to track remote branch 'change1' from 'origin'.
```

The continuous integration should be triggered as soon as PR is created and fails with the following error message:

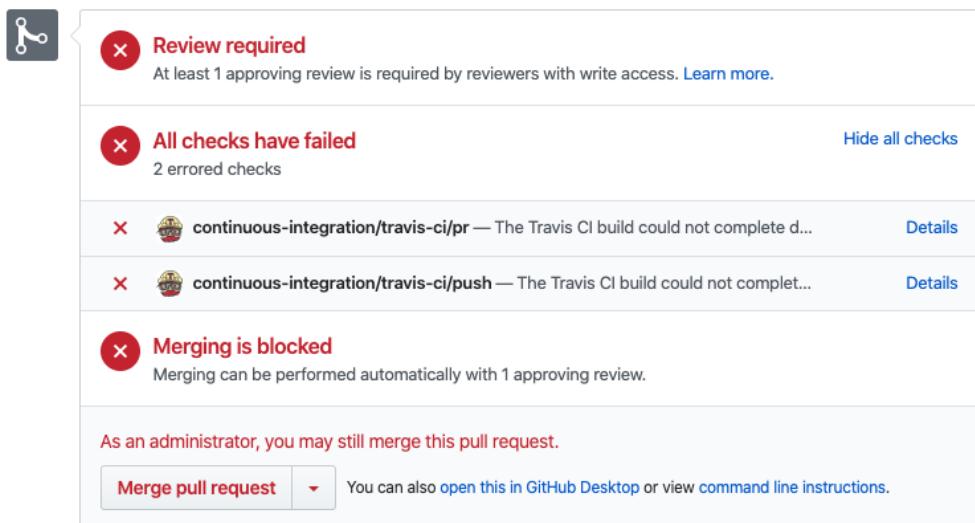


Figure 6.5 Travis runs the CI job that validates deployment manifests. The validation fails due to detected vulnerabilities.

```
time="2019-12-17T09:05:41Z" level=error msg="RunAsNonRoot is not set in
ContainerSecurityContext, which results in root user being allowed!
Container=sample-app KubeType=deployment Name=sample-app"
```

The Kubeaudit detected that the pod security context is missing the `runAsNonRoot` property that prevents running a container with 'root' user as part of the pod. This is a valid security concern. To fix the security issue, change pod manifest as represented at code listing below.

Listing 6.3 Sample App Deployment ([deployment.yaml](#)).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - image: gitopsbook/sample-app:v0.1
          name: sample-app
          command:
            - /app/sample-app
          ports:
            - containerPort: 8080
+        securityContext:
+          runAsNonRoot: true
```

Commit changes and update the pull request by pushing '`change1`' branch.

```
git commit -am 'Update deployment'
git push upstream change1
```

The pull request should successfully pass verification!

Exercise 6.4

Learn which additional audits are provided by the `kubeaudit` application. Try using `kubeaudit autofix -f deployment.yaml` command.

PROTECT COMMIT AUTHOR IDENTITY

At this point, our repository is securely hosted on Github. We control which Github accounts can make changes in the repository, enforce the code review process for every change, and even run static analysis for every pull request. This is great but still not enough. As it often happens, a social engineering attack can bypass all these security gates.

What would you do if your boss sends you a pull request and asks you to merge it immediately? Under pressure, an engineer might decide to give a quick look at the pull request and approve it without careful testing. Since our repository is hosted on Github, we

know which user authored the commit. It is impossible to make a commit on behalf of someone else, right?

Unfortunately, this is not true. Git was not designed with strong identity guarantees. As we mentioned before, Git is an extremely developer-oriented tool. Every bit of a commit is under engineers' control, including information about the commit author. So an intruder can easily create a commit and put your boss's name into the commit metadata. Let's do a simple exercise to demonstrate this vulnerability.

Open a console and create a new commit on the master branch using the command below:

```
echo '# hacked' >> ./deployment.yaml
git commit --author='Joe Beda <joe.github@bedafamily.com>' -am 'evil commit'
git push upstream master
```

Open the commit history of your repository on Github and check the most recent commit information. Look, Joe Beda⁴⁹ just updated our pod manifest!

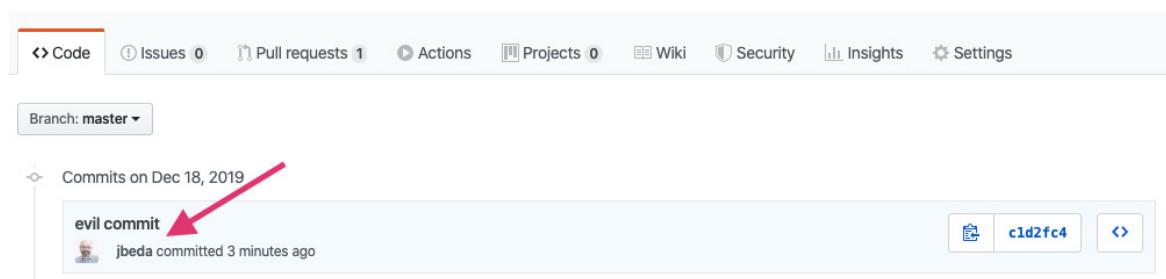


Figure 6.6 Github commit history contains the Joe Beda avatar. By default, Github does not execute any validation and just uses author information stored in the commit metadata.

That looks pretty scary, but it does not mean that going forward, you need to personally verify the identity of every pull request author before approving it. Instead of manually verifying who is the commit's author, you might leverage a digital crypto signature.

The cryptographic tools like GPG allow you to inject a crypto signature into the commit metadata. Later this signature might be verified by the Git hosting service or GitOps operator. It would take too much time to learn exactly how a GPG signature works, but we definitely can use it to protect our deployment repo.

Unfortunately, the GPG configuration process might be difficult. It includes multiple steps which might vary depending on your operating system. Please refer to steps described in Appendix C and the Github online documentation⁵⁰ to configure the GPG key.

Finally, we are ready to make a commit and sign it. The command below creates a new change to the deployment manifest and signs it with the GPG key associated with your Github account:

⁴⁹ Joe Beda - one of the principal founders of Kubernetes. <https://www.linkedin.com/in/jbeda>

⁵⁰ <https://help.github.com/en/github/authenticating-to-github/adding-a-new-gpg-key-to-your-github-account>

```
echo '# signed change' >> ./deployment.yaml
git add .
git commit -S -am 'good commit'
git push upstream master
```

Now the Github commit history includes information about the author that is based on a GPG key and cannot be faked.



Figure 6.7 Signed Git cryptographically protects author identity. The Github user interface visualizes GPG verification results.

Github allows you to require that all commits to a particular repository be signed. The "Require signed commits" setting is available under the "Protected branches" section of the repository settings.

In addition to Git hosting service confirmation, you might configure your GitOps operator to verify GPG signatures automatically before updating the Kubernetes cluster configuration. Fortunately, some GitOps operators have built-in signature verification support and don't require complex configuration. This topic will be covered in the following chapters.

6.2.2 Kubernetes RBAC

As you already know, GitOps methodology assumes that a continuous integration pipeline has no access to the Kubernetes cluster. The only automation tool with direct cluster access is the GitOps operator that lives inside the cluster. This is already a great advantage over the traditional DevOps model. However, that does not mean that GitOps should have god-level access. We still need to think carefully about which permissions level the operator should get. The operator inside of a cluster, the so-called pull model, is not the only possibility either. You might consider placing the GitOps operator inside of the protected perimeter and reduce management overhead by using a push-model and managing multiple clusters using one operator. Each such consideration has some pros and cons. To make a meaningful decision, you need to have a good understanding of the Kubernetes access model. So let's step back and learn which security tools are built-in into Kubernetes and how we can use them.

ACCESS CONTROL TYPES

There are four well-known access control flavors:

- **Discretionary access control (DAC)** With DAC models, the data owner decides on

access. DAC is a means of assigning access rights based on rules that users specify.

- **Mandatory access control (MAC)** MAC was developed using a nondiscretionary model, in which people are granted access based on an information clearance. MAC is a policy in which access rights are assigned based on regulations from a central authority.
- **Role-Based Access Control (RBAC)** RBAC grants access based on a user's role and implement key security principles, such as "least privilege" and "separation of privilege." Thus, someone attempting to access information can only access data that are deemed necessary for their role.
- **Attribute-Based Access Control (ABAC)** Attribute-based access control (ABAC), also known as policy-based access control, defines an access control paradigm whereby access rights are granted to users through the use of policies that combine attributes together.

The ABAC is very flexible and probably the most powerful model from the list. Because of its power, the ABAC was initially chosen as a Kubernetes security model. However, later on, the community has realized that ABAC concepts and the way they were implemented in Kubernetes are hard to understand and use. As a result, the new authorization mechanism that is based on RBAC was introduced. In 2017 the RBAC based authorization was moved into beta, and ABAC had been declared as deprecated. Currently, RBAC is the preferred authorization mechanism in Kubernetes and recommended to be used for every application running on Kubernetes.

The RBAC model includes the following three main elements: subjects, resources, and verbs. Subjects represent users or processes that want to access a resource, and the verb is an operation that can be executed against a resource.

So how are these elements mapped to Kubernetes API objects? The RBAC resources are represented by a regular Kubernetes resource such as Pod or Deployment. In order to represent verbs, two new sets of specialized resources were introduced in Kubernetes. The verbs are represented by Role, and RoleBinding resources and subjects are represented by User and ServiceAccount resources.

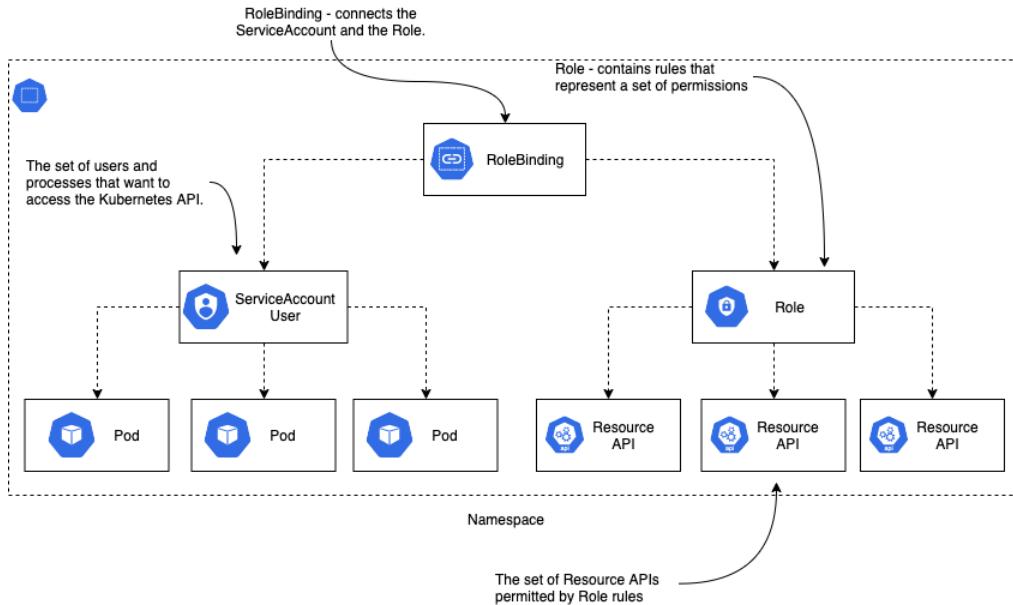


Figure 6.8 Kubernetes RoleBinding grants permissions defined in the Role to Users and ServicesAccounts. ServiceAccount provides an identity for processes that run in a Pod.

ROLE AND ROLEBINDING

Role resource is meant to connect verbs and Kubernetes resources. The sample role definition is represented at the code listing below:

Listing 6.4 Sample Role ([role.yaml](#)).

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: sample-role
rules:
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
  - list
```

The verbs section contains lists of allowed action. So altogether that role allows listing config maps with the cluster and get the detailed information of every config map. The advantage of a role is that it is a reusable object and can be used for different subjects. For example, you might define the `read-only` role and assign it to the various subjects without duplicating the resources and verbs definitions.

It is important to know that Roles are namespaced resources and provide access to the resources defined in the same namespace⁵¹. This means that a single role cannot provide access to the resources in multiple namespaces or cluster level resources. In order to provide the cluster level access, you might use an equivalent resource called ClusterRole. The ClusterRole resource has pretty much the same set of fields as a Role, with the exception of the namespace field.

The RoleBinding enables the Role to be connected with the subjects. A sample RoleBinding definition is represented below:

Listing 6.5 Sample Role Binding ([role-binding.yaml](#)).

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sample-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sample-role
subjects:
- kind: ServiceAccount
  name: sample-service-account
```

The represented sample RoleBinding grants a set of permissions defined in the Role named `sample-role` to the ServiceAccount named `sample-service-account`. Similarly to the Role, a RoleBinding has an equivalent object, ClusterRoleBinding, that allows connecting subjects with the ClusterRole.

USER AND SERVICEACCOUNT

Finally, Kubernetes subjects are represented by the Service Accounts and Users.

BASIC GITOPS OPERATOR RBAC

We have already configured Kubernetes RBAC in Chapter 2 while working on the basic GitOps operator implementation. Let's use the new knowledge acquired in this chapter to tighten the operator permissions and limit what it can deploy.

To get started, make sure you've completed the basic GitOps operator tutorial. As you might remember, we have configured a CronJob along with ServiceAccount and the ClusterRoleBinding resources. Let's take a look at ServiceAccount and ClusterRoleBinding definition one more time and find what should be changed to improve security:

Listing 6.6 rbac.yaml.

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitops-serviceaccount
  namespace: gitops
```

⁵¹ <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: gitops-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: admin
subjects:
- kind: ServiceAccount
  name: gitops-serviceaccount
  namespace: gitops
```

The ClusterRoleBinding defines the link between ClusterRole name “admin” and ServiceAccount that is used by the GitOps operator CronJob. The “admin” ClusterRole exists by default in the cluster and provides god-level access to the whole cluster. That means that the GitOps operator has no limitations and can deploy any resource as long as it is defined in the Git repository.

So what is wrong in this RBAC configuration? The problem is that this is secure only if we assume that the developer with the Git repository write permissions already has full cluster access. Since the GitOps operator can create any resource, the developer might add manifests of additional roles and role bindings and grant himself admin permissions. This is not what we want, especially in a multi-tenant environment.

Another consideration is human mistakes. When the cluster is used by multiple teams, we need to ensure that one team cannot touch the resources of another team. As you learned from Chapter 3, the teams are typically separated from each other using Kubernetes namespaces. So it makes sense to limit GitOps operator permissions to one namespace only.

Finally, we want to control what namespace level resources can be managed by the GitOps operator. While it is perfectly fine to let developers manage such resources like Deployments, ConfigMaps, and Secrets, there are some resources that should be managed by the cluster administrator only. A good example of restricted network resources is NetworkPolicy. The NetworkPolicy controls what traffic is allowed to the pods within the namespace and typically managed by cluster administrators.

Let's go ahead and update the RBAC configuration of the operator. We would have to make the following changes to ensure secure configuration:

- Limit GitOps operator permissions to only one namespace
- Remove permissions to install cluster level resources
- Limit operator permissions to selected namespaced resources

The updated RBAC configuration is represented below:

Listing 6.7 updated-rbac.yaml.

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: gitops-serviceaccount
  namespace: gitops
```

```

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: gitops-role
  namespace: gitops
rules:
- apiGroups:
  - ""
    resources:
    - secrets
    - configmaps
    verbs:
    - "*"
- apiGroups:
  - "extensions"
  - "apps"
    resources:
    - deployments
    - statefulsets
    verbs:
    - "*"

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: gitops-role-binding
  namespace: gitops
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: gitops-role
subjects:
- kind: ServiceAccount
  name: gitops-serviceaccount

```

Here is a summary of applied changes:

- The ClusterRoleBinding was replaced with the RoleBinding to ensure Namespace level access only.
- Instead of using a built-in admin role, we are using the custom namespace Role.
- The namespaced role provides access only to specified Kubernetes resources. That ensures that the operator cannot modify resources like NetworkPolicy.

6.2.3 Image Registry Access

By securing the Kubernetes cluster, we guarantee that cluster configuration describes the correct workloads that reference that correct docker images and ultimately run the software which we want. The protected deployment repository and fully automated GitOps driven deployment process provides audibility and observability. The last missing piece that is still not protected is the Docker image itself.

The Docker image protection topic is listed last, but this is definitely not the least important topic. The image content ultimately defines what binary is going to be executed

inside of a cluster. So even if everything else is secure, the breach in Docker registry protection defeats all other security gates.

So what does docker image protection mean in practice? We would have to take care of the two following issues:

- The registry images cannot be changed without permission
- Images are securely delivered into the Kubernetes cluster.

REGISTRY IMAGE PROTECTION

Similarly to the Git repository, the Docker repository protection is provided by the hosting service. Probably the most popular docker repository hosting service is the Docker Hub. The service allows accessing thousands of docker images. The service is provided by Docker Inc and completely free for any open-source project.

To get hands-on experience with Docker Hub, you need to get an account on Docker Hub, create a repository, and push an image. Unless you already have an account, navigate to <https://hub.docker.com/signup> and create one. As a next step, we need to create a docker repository named gitops-k8s-security-alpine, as described in Docker Hub documentation⁵². Finally, we are ready to verify if Docker Hub is protecting the repository but before we need to get the sample Docker image. The simplest way to create one is to pull an existing image and rename it. The following command pulls the Alpine Linux image from the official Docker Hub repository and renames it to <username>/gitops-k8s-security-alpine where <username> is the name of your Docker Hub account:

```
docker pull alpine
docker tag alpine <username>/gitops-k8s-security-alpine:v0.1
```

The next command pushes the image into the gitops-k8s-security-alpine docker registry:

```
docker push <username>/gitops-k8s-security-alpine:v0.1
```

However, the local docker client does not have credentials to access the Docker Hub repository, so the push command should fail. To fix the error, run the command below and provide your Docker Hub account username and password:

```
docker login
```

Once you have logged in successfully, the Docker client knows who you are, and the docker push command can be executed.

SECURING IMAGE DELIVERY

The security of image delivery into the cluster means answering the question: "do we trust the source of the image?". And trust means that we want to be sure that the image was created by the authorized author, and the image content is not being modified while transferring from the repository. So this is the problem of protecting the image author identity. And the solution is very similar to the solution that protects Git commit author identity:

⁵² <https://docs.docker.com/docker-hub/repos/>

- The person or an automated process uses the digital signature to sign the content of the image.
- The signature is used by the consumer to verify that the image was created by the trusted author, and the content was not tampered with.

The good news is that this is already supported by the docker client and the image registry. The docker feature named Content Trust allows signing the image and push it into the registry along with the signature. The consumer can use the Content Trust feature to verify that the signed image content was not changed.

So in the perfect scenario, the CI pipeline should publish the signed image, and Kubernetes should be configured to require a valid signature for every image running in production. The bad news is that Kubernetes, as of version v1.17, still does not provide the configuration that enforces image signature verification. So the best we can do is verify the image signature before modifying the Kubernetes manifests.

The Content trust configuration is fairly simple. You have to set the **DOCKER_CONTENT_TRUST** environment variable:

```
export DOCKER_CONTENT_TRUST=1
```

Once the environment variable is set, the Docker commands `run` and `pull` should verify the image signature. We can confirm it by pulling the unsigned image that we just pushed to the “gitops-k8s-security-alpine” repository:

```
$ docker pull <username>/gitops-k8s-security-alpine:v0.1
Error: remote trust data does not exist for docker.io/<username>/gitops-k8s-security-
alpine: notary.docker.io does not have trust data for docker.io/<username>/gitops-
k8s-security-alpine
```

The command fails as expected because “`<username>/gitops-k8s-security-alpine:v0.1`” image was not signed. Let’s fix it. Make sure the `DOCKER_CONTENT_TRUST` environment variable are still set to “1” and create signed image using following command:

```
$ docker tag alpine <username>/gitops-k8s-security-alpine:v0.2
$ docker push <username>/gitops-k8s-security-alpine:v0.2
The push refers to repository [docker.io/<username>/gitops-k8s-security-alpine]
6b27de954cca: Layer already exists
v0.2: digest: sha256:3983cc12fb9dc20a009340149e382a18de6a8261b0ac0e8f5fcdf11f8dd5937e size:
528
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID cfe0184:
Repeat passphrase for new root key with ID cfe0184:
Enter passphrase for new repository key with ID c7eba93:
Repeat passphrase for new repository key with ID c7eba93:
Finished initializing "docker.io/<username>/gitops-k8s-security-alpine"
Successfully signed docker.io/<username>/gitops-k8s-security-alpine:v0.2
```

This time `docker push` command signs the image before pushing it. If you are pushing the signed image for the first time, Docker will generate keys in the `~/.docker/trust/` directory and prompt you for the passphrase to use for the root key and repo key. After providing the passphrase, the signed image is pushed to the docker hub. Finally, we can verify that the pushed image has a proper signature by running `docker pull` command one more time:

```
docker pull <username>/gitops-k8s-security-alpine:v0.2
```

This time command successfully completed that proof that our image has proper signature and the docker client was able to verify it!

6.3 Patterns

OK, let's face it. Brand-new greenfield projects are not necessarily started with a perfectly secure deployment process. In fact, young projects don't even have an automated deployment process. The lead engineer may be the only one able to deploy the project, which she may do from her laptop. Typically, a team starts adding automation as it gets more and more time consuming to deploy all the application services. As the potential cost and damage from unauthorized access increases, the security of that automation becomes more and more critical.

6.3.1 Full access

The initial security model of almost every new project is usually based completely on trust between the team members. Every team member has full access, and deployment changes are not necessarily recorded and available for an audit later.

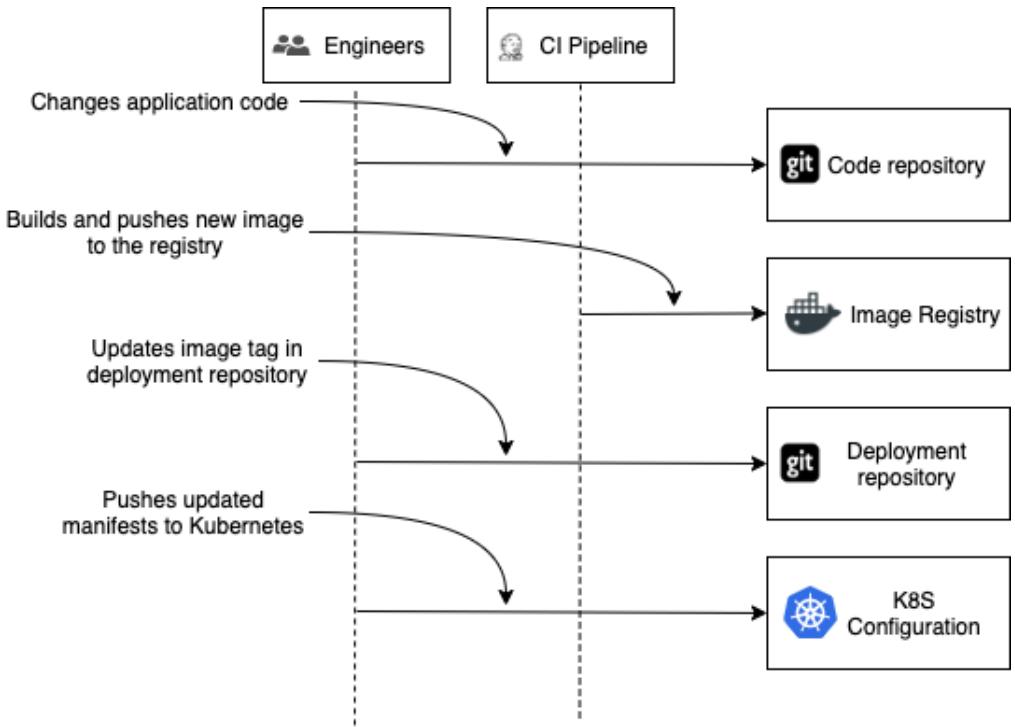


Figure 6.9 The full access security model assumes that both engineers and continuous integration systems have full access to the Kubernetes cluster. The trade-off is speed over security and more applicable for new projects in the beginning stage.

Weak security is not necessarily a bad thing in the beginning. Full access means fewer barriers that enable the team to be more flexible and move more quickly. While there is no important customer data in production, it is a perfect chance to focus on speed and shape the project until you are ready to move into production. But probably sooner rather than later, you will need to put proper security controls in place, not only for customer data in production but also to ensure the integrity of the code being deployed into production.

6.3.2 Deployment Repo Access

Disabling direct Kubernetes access for developers by default is a big step forward from a security perspective. This is the most common pattern if you are using GitOps. In this model, developers still have full access to the deployment repository but must rely on the GitOps operator to push changes to the Kubernetes cluster.

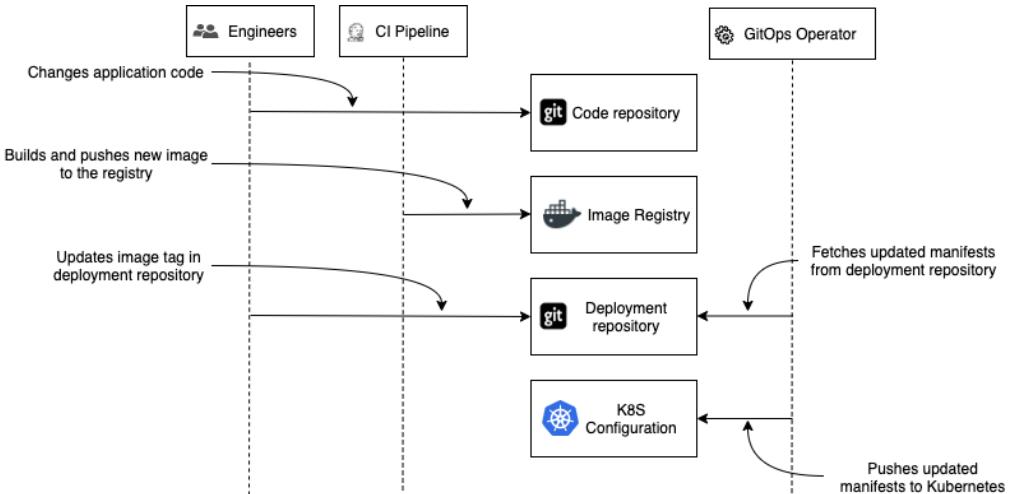


Figure 6.10 GitOps operator allows removing cluster access. At this point, engineers need access only to the deployment repository.

In addition to better security, this pattern provides auditability. Assuming that no one has access to the Kubernetes configuration, the deployment repository history contains all cluster configuration changes.

The pattern is still not perfect. While the project is maturing and the team keeps improving the deployment configuration, it feels perfectly fine to update the deployment repository manually. However, after some time, each application release is going to require only the image tag change. At this stage, the maintenance of the deployment repository is still very valuable but may feel like a lot of overhead.

6.3.3 Code access only

The "code access only" pattern is a logical continuation of deployment "repository access only." If the release changes in the deployment repository are predictable, then it is possible to codify the configuration change process in the CI pipeline.

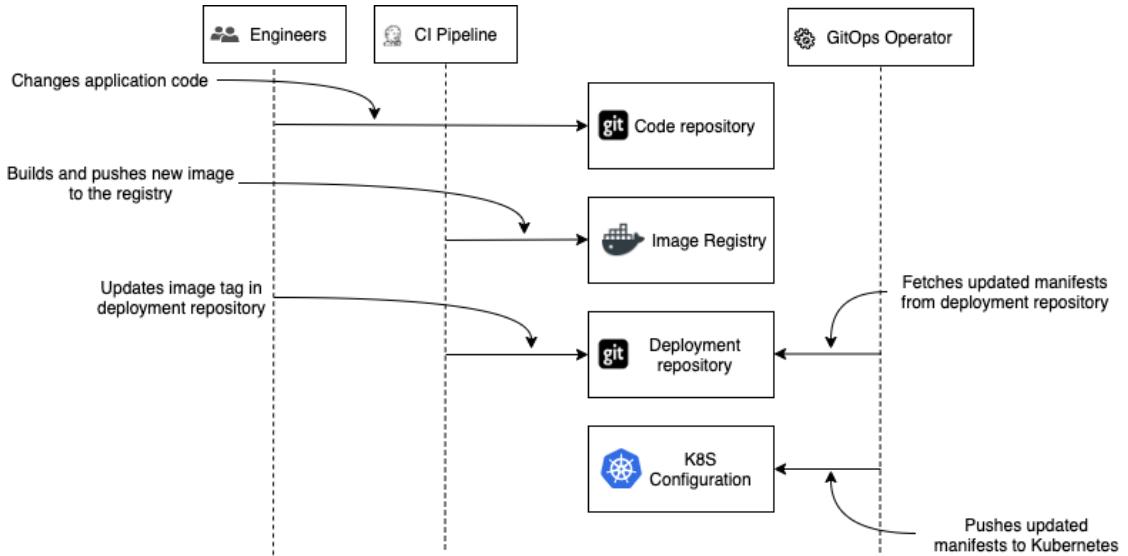


Figure 6.11 Code access only mode assumes that both the deployment repository and Kubernetes cluster changes are fully automated. Engineers need only code repository access.

The pattern streamlines the development process and significantly reduces the amount of manual work. It also improves the deployment security in several ways:

- The development team no longer needs access to the deployment repository. Only the dedicated automation account has permission to push into the repository.
- Since changes in the deployment repository are automated, it is much easier to configure the GPG signing process and automate it in the CI pipeline.

Exercise 6.5

Choose which pattern fits your project the most. Try to elaborate on the pros and cons of each pattern and explain why you would prefer the chosen pattern.

6.4 Security Concerns

We've learned how to protect our deployment process end-to-end, starting from the most basic all the way to identity protection of configuration changes and new images. Finally, let's learn important edge cases that must be covered to ensure the security of your cluster.

6.4.1 Prevent Image Pull from Untrusted Registries

In Section 6.2.3, we show how you can implement security controls on public registries, such as docker.io, to ensure the images have been published by authorized users as intended, and have not been tampered with when being pulled. However, the fact is that a public registry is outside of your visibility and control. You must trust that the maintainers of the public

registry are following security best-practices. And even if they are, the fact that they are a public registry means that anyone on the Internet can push images to it. For some businesses with very high-security needs, this is not acceptable.

To address this, many enterprises will maintain their own private Docker image registry for reliability, performance, privacy, and security. In this case, new images should be pushed to the private registry (e.g., **docker.mycompany.com**) instead of a public registry (e.g., **docker.io**). This can be accomplished by modifying the CI pipeline to push successfully built new images to the private registry.

Deployments to Kubernetes should also only pull images from the private registry. But how can this be enforced? What if a naive developer accidentally pulls a virus- or malware-infected image from docker.io? Or a malicious developer who doesn't have privileges to push images to the private registry but tries to "side-load" an image from their public Docker Hub repository? Of course, using GitOps will ensure that these actions are recorded in the audit trail so those responsible should be able to be identified. However, how can this be prevented in the first place?

This can be accomplished using the Open Policy Agent (OPA), and an admission webhook that rejects manifests that reference an image coming from a prohibited image registry.

6.4.2 Cluster Level Resources in Git Repository

As you know from this chapter, the Kubernetes access settings are controlled using Kubernetes resources such as Role and ClusterRole. RBAC resources management is a perfectly valid use of a GitOps operator. It is common practice to package the definition of an application deployment together with required Kubernetes access settings. However, there is a potential security hole that can be used to escalate privileges. Because Kubernetes access settings are managed by resources, these resources can be placed into the deployment repository and delivered by the GitOps operator. The intruder might create a ClusterRole and give permissions to the service account that later on might be used as a back door.

The rule of thumb that prevents the privileges escalation is to limit the GitOps operator privileges itself. If the development team that leverages the GitOps operator is not supposed to manage ClusterRoles, then the GitOps operator should not have that permission. If the GitOps operator is shared by multiple teams, the operator should be configured appropriately and should enforce the team-specific security checks.

Exercise 6.6

Refer back to the poor man's GitOps operator tutorial. Review the RBAC configuration and check if it allows the security privileges escalation attack.

6.5 Summary

- Traditional CI Ops security model has a wide attack surface due to access by both engineers and continuous integration systems. GitOps security model significantly reduces the attack surface of the cluster because only the GitOps operator has access to the cluster.
- Git's underlying data structure uses Merkle tree which provides a tree-like structure with cryptographic protection to provide tamper proof commit log.

- In addition to the Git's data structure security advantage, code review process using Pull Request and automated checks using tools such as kubeaudit and kubesec can detect security vulnerability in the manifests.
- Git natively does not protect commit author identity. Using GPG can guarantee the authenticity of the commit author by injecting a digital crypto signature in the commit metadata.
- Role-Based Access Control (RBAC) is the preferable way to implement Access Control in Kubernetes. Both users and GitOps operators' access control can be provisioned through RBAC.
- Similar to Git, all docker images should be signed with digital signature to verify the authenticity using the Content Trust feature.
- New projects can start with full access (cluster, deployment repo and code repo) for engineers to focus on development velocity initially. As the projects mature and get ready for initial production release, both cluster and deployment repo access should be restricted to put high emphasis on security over speed.

7

Secrets

This chapter covers:

- Kubernetes Secrets
- GitOps strategies for managing secrets
- Tooling for managing Secrets

Kubernetes provides a mechanism allowing users to store small bits of sensitive information into a protected resource object, called a Secret. A Secret is anything that you want to tightly control access to. Common examples of data which you would want to store in a Secret include things like username and password credentials, API keys, SSH keys, TLS certificates. In this chapter you will learn about different Secret management strategies when using a GitOps system. You will also have a brief introduction to several different tools that can be used for storing and managing secrets.

We recommend you read chapters 1 and 2 before reading this chapter.

7.1 Kubernetes Secrets

A simple *Kubernetes Secret* is a data structure composed of three pieces of information:

- The name of the Secret
- The type of the secret (optional)
- A map of field names to sensitive data, encoded in base64

A very basic secret looks like the following:

Listing 7.1 example-secret.yaml.

```
apiVersion: v1
kind: Secret
metadata:
  name: login-credentials
```

```

type: Opaque          #A
data:
  username: YWRtaW4=  #B
  password: UEA1NXcwcwcmQ=  #C

```

#A Type of the secret - used to facilitate programmatic handling of secret data
#B The string "admin" base64 encoded
#C The string "P@55w0rd" base64 encoded

When looking at the values of a Secret for the first time, at first glance you might mistakenly think that the Secret values were protected with encryption, since the fields are not readable by a human, and are not presented in plain-text. But you would be mistaken, and it is important to understand that:

- Secret values are base64 encoded
- Base64 encoding is **not** the same thing as encryption
- Viewing should be considered the same as plain text.

NOTE **Base64 Encoding** Base64 is an encoding algorithm that allows you to transform any characters into an alphabet which consists of Latin letters, digits, plus, and slash. It allows binary data to be represented in an ASCII string format. Base64 does not provide encryption.

The reason Kubernetes base64 encodes the data at all, is that it allows Secrets to store binary data. This is important for storing things like certificates as secrets. Without base64 encoding, it would be impossible to store binary configurations as a Secret.

7.1.1 Why Secrets?

Using Secrets is optional in Kubernetes, but more convenient, flexible, and secure than other techniques, such as placing the sensitive values directly in the pod specification, or baking the values into the container image during build time.

Just as with ConfigMaps, Secrets allow the separation of the configuration of an application from the build artifact.

7.1.2 How to use Secret?

Kubernetes Secrets, just like ConfigMaps, can be used in several ways:

1. As files mounted as files in the Pod
2. As environment variable in the Pod
3. Kubernetes API access

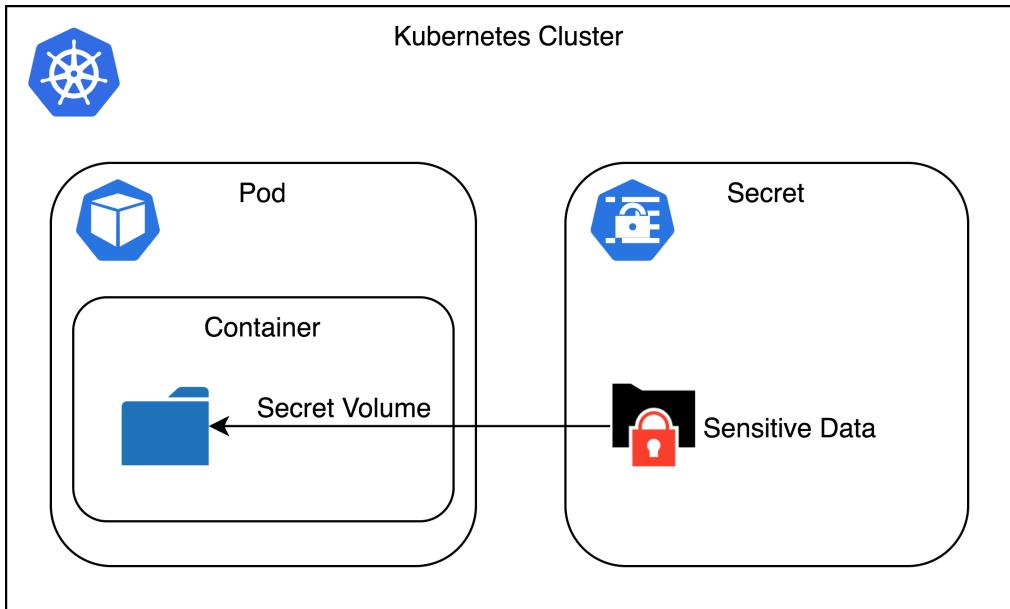


Figure 7.1 Secret Volume - A secret volume is used to pass sensitive information, such as passwords, to Pods. Secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

VOLUME MOUNTING SECRETS AS FILES IN A POD

The first technique of utilizing secrets is by mounting them into a Pod as a volume. To do this, you first declare

Listing 7.2 secret-volume.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  Volumes:                      #A
  - name: foo
    secret:
      secretName: mysecret
  containers:
  - name: mycontainer
    image: redis
    volumeMounts:          #B
    - name: foo
      mountPath: /etc/foo
      readOnly: true
```

#A A volume of type Secret is declared in the pod, with an arbitrary name

#B The container that needs the secrets specifies a path of where to mount the secret data volume

When projecting a Secret (or configmap) into a Pod as a volume of files, changes to the underlying Secret will eventually update the files mounted in the Pod. This allows the opportunity for the application to reconfigure itself, or "hot-reload", without a restart of the container/pod.

USING SECRETS AS ENVIRONMENT VARIABLES

The second way of utilizing Kubernetes Secrets, is by setting them as environment variables.

Listing 7.3 secret-environment-variable.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: mycontainer
      image: redis
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret      #A
              key: username       #B
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysecret      #A
              key: password        #B
```

#A Name of the secret

#B The key of the secret data map

Exposing Secrets to containers as environment variables, while convenient, is arguably not the optimal way to consume secrets, since it is less secure than consuming it as a volume mounted file. When a Secret is set as an environment variable, all processes in the container (including child processes) will inherit the OS environment, and be able to read the environment variable values, and thus to the secret data. For example, a forked shell script would be able to read the environment variables by running the `env` utility.

Disadvantage of secret environment variables: A second disadvantage of using Secrets as environment variables, is that unlike Secrets projected into volumes, values of secret environment variables will not be updated if the Secret is ever updated after the container starts. A container or pod restart would be necessary to notice changes.

USING SECRETS FROM THE K8S API

Finally, Kubernetes Secrets can also be retrieved directly from the Kubernetes API. Suppose you had the following Secret with a password field:

Listing 7.4 secret.yaml.

```
apiVersion: v1
kind: Secret
```

```
metadata:
  name: my-secret
type: Opaque
data:
  password: UEA1NXcwcMQ=
```

To retrieve the secret, the Pod itself could retrieve the secret value directly from Kubernetes for example using a `kubectl` command, or REST API call. The following `kubectl` command retrieves the Secret named `my-secret`, base64 decodes the password field, and prints the plain-text value to standard out.

```
$ kubectl get secret my-secret -o=jsonpath='{.data.password}' | base64 --decode
P@55w0rd
```

This technique requires that the Pod has privileges to retrieve the secret.

SECRET TYPES

The secret “type” field is an indication of what type of data is contained inside the secret. It is primarily used by software programs, to identify relevant secrets it might be interested in, as well as safely make assumptions about what available fields inside the secret are set.

The following table describes the built-in Kubernetes Secret types, as well as the required fields for that type.

Table 1. (Built-in Secret Types)

Type	Description	Required Fields
Opaque	The default type. Contains arbitrary user-defined data.	
kubernetes.io/service-account-token	Contains a token that identifies a service account to the Kubernetes API.	data["token"]
kubernetes.io/dockercfg	Contains a serialized <code>~/.dockercfg</code> file.	data[".dockercfg"]
kubernetes.io/dockerconfigjson	Contains a serialized <code>~/.docker/config.json</code> file.	data[".dockerconfigjson"]
kubernetes.io/basic-auth	Contains basic username/password credentials.	data["username"] data["password"]

kubernetes.io/ssh-auth	Contains a private SSH key needed for authentication.	data["ssh-privatekey"]
kubernetes.io/tls	Contains a TLS private key and certificate.	data["tls.key"] data["tls.crt"]

7.2 GitOps and Secrets

Kubernetes GitOps practitioners will invariably come to the same problem: while users are perfectly comfortable with their storing configuration into git, when it comes to sensitive data, they are unwilling to store their data in git due to security concerns. Git was designed as a collaborative tool, making it easy for multiple people and teams to gain access to code and view each other's changes. But these same properties are also what cause the use of git to hold secrets, extremely dangerous. There are many concerns and reasons why it is inappropriate to store Secrets in git, which we cover below.

No ENCRYPTION

As we learned earlier, Kubernetes provides no encryption on the contents of a Secret, and the base64 encoding of the values should be considered the same as plain-text. Additionally, git alone does not provide any form of built-in encryption. So when storing secrets in a git repository, the secrets are laid bare to anyone with access to the git repository.

DISTRIBUTED GIT REPOS

With GitOps, you and your colleagues will be locally cloning the git repository to your laptops and workstations, for the purposes of managing the configuration of the applications. But by doing so, you would also be proliferating and distributing secrets to many systems, without adequate auditing or tracking. If any of these systems were to become compromised (e.g. hacked or even physically lost), then someone would gain access to all of your secrets.

NO GRANULAR (FILE-LEVEL) ACCESS CONTROL

Git does not provide read protection of a subpath, or sub file of a git repository. In other words, it is not possible to restrict access to some files in the git repository, but not others. When dealing with secrets, you ideally should be granting read access to the secrets on a need-to-know basis. For example, if you had a temporary worker who needed partial access to the git repository, you would want to give the least amount of access to the content as possible to that user. Unfortunately, git does not provide any facilities to accomplish this and it is an all-or-nothing decision when giving permissions to a repo.

INSECURE STORAGE

Git was never intended to be used in the capacity of a secret management system. As a result, it did not design into the system standard security features such encryption at rest. Therefore, a compromised git server would have the potential of also leaking all secrets of all the repositories it manages, making it a prime target for attack.

NOTE *Git Provider Features* Although git in itself does not provide security features such as encryption-at-rest, *git providers* often do provide these features *on top of git*. For example, GitHub does claim to encrypt repositories at rest. But this functionality may vary from provider-to-provider.

FULL COMMIT HISTORY

Once a secret is added to the Git commit history it is very difficult to remove it. If the secret is checked into Git and then later deleted, that secret can still be retrieved by checking out an earlier point in the repository history before the secret was deleted. Even if the secret is encrypted, when the key used to encrypt the secret is later rotated and the secret is re-encrypted with a new key, the secret encrypted with the old keys are still present in the repository history.

7.3 Secret Management Strategies

There are many different strategies of dealing with Secrets in GitOps with tradeoffs in flexibility, manageability, and security. Before going into the application of tools to implement these strategies, we'll first go over some of these strategies at a conceptual level.

7.3.1 Store Secrets in Git

The first "strategy" of GitOps and Secrets, is to not have a strategy at all. In other words, you would simply commit and manage your Secrets in git like any other Kubernetes resource, and accept the security consequences.

You might be thinking: "what is so wrong about storing my secrets in git?" Even if you have a private GitHub repository, which is only accessible by trusted members of your team, you might want to allow 3rd party access to the git repo. For example, CI/CD systems, security scanners, static analysis, etc... By providing your git repository of secrets to these 3rd party software systems, you are in turn entrusting them with your secrets.

So in practice, the only real acceptable scenarios where Secrets could be stored as-is in git, are when the secrets do not contain any truly sensitive data, such as dev and test environments.

7.3.2 Bake Secrets into the Container Image

One naive strategy that might come to mind to avoid storing Secrets in git, is to "bake" the sensitive data directly into the container image. In this approach, the secret data is directly copied into the container image as part of the docker build process.

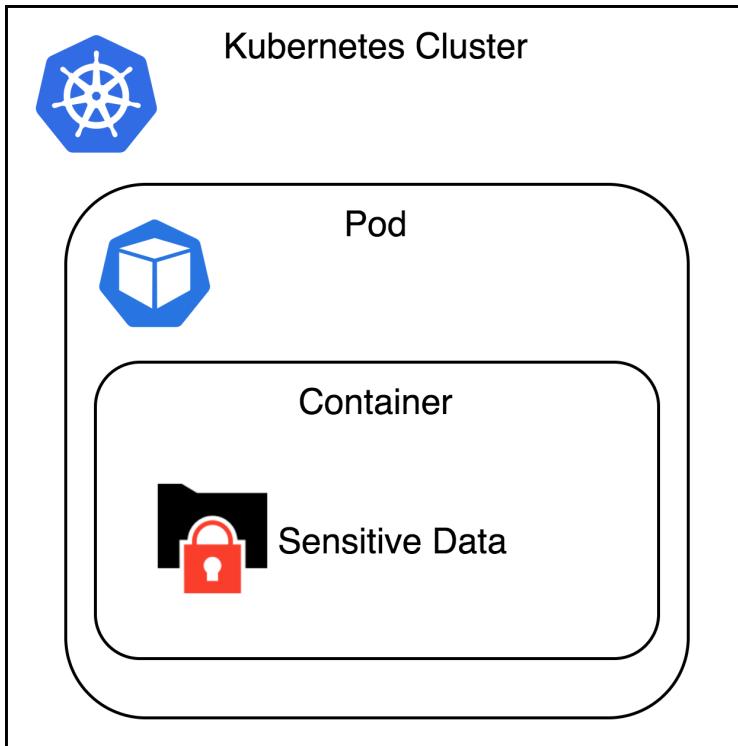


Figure 7.2 Baking a secret into the container image - The docker build process “bakes in” the sensitive data into the image (e.g. by copying the sensitive file into the container). No secret store is used (Kubernetes or external), but the container registry becomes sensitive since it is effectively a secret store.

A simplistic Dockerfile which bakes in secrets into the image might look like:

Listing 7.5 Dockerfile-with-secret.

```
FROM scratch

COPY ./my-app /my-app
COPY ./credentials.txt /credentials.txt

ENTRYPOINT ["/my-app"]
```

Advantages of this approach are that this approach removes git, and even Kubernetes itself from the equation. In fact, with the secret data baked into the container image, the image can be run anywhere, not just Kubernetes, and work without any configuration.

However, baking the sensitive data directly into the container image has some very bad drawbacks, which should automatically rule it out as a viable option.

The first issue is that the container image *itself* is now sensitive. Due to the fact that the sensitive data was copied into the image, anyone or anything that has access to the container image (e.g. via a `docker pull`), can now trivially copy out and retrieve the secret.

Another problem is that because the secret is baked into the image, it makes updates to the secret data extremely burdensome. Whenever the credentials need to be rotated, it would require a complete rebuild of the container image.

A third problem is that the container image is not flexible enough to accomodate when the same image needs to run using different secret datasets. Suppose you had three environments where this container image will be deployed:

1. A developer environment
2. A test environment
3. A production environment

Each of these environments needs a different set of credentials because it connects to three different databases. The approach of baking in the secret data into the container image would not work here, because it can only choose one of the database credentials to bake into the image.

7.3.3 Out-of-Band Management

A second approach for dealing with secrets in GitOps, is to manage Secrets completely *out-of-band* from GitOps. With this approach, everything *except* Kubernetes Secrets would be defined in git and deployed via GitOps, but some other mechanism would be used for deploying Secrets, even if it was manual.

For example, a user could store their secrets in a database, a cloud provider's managed secret store, even a text file on their local workstation. When it comes time to deploy, the user would manually run `kubectl apply` to deploy the Secret into the cluster, and then let a GitOps operator to deploy everything else.

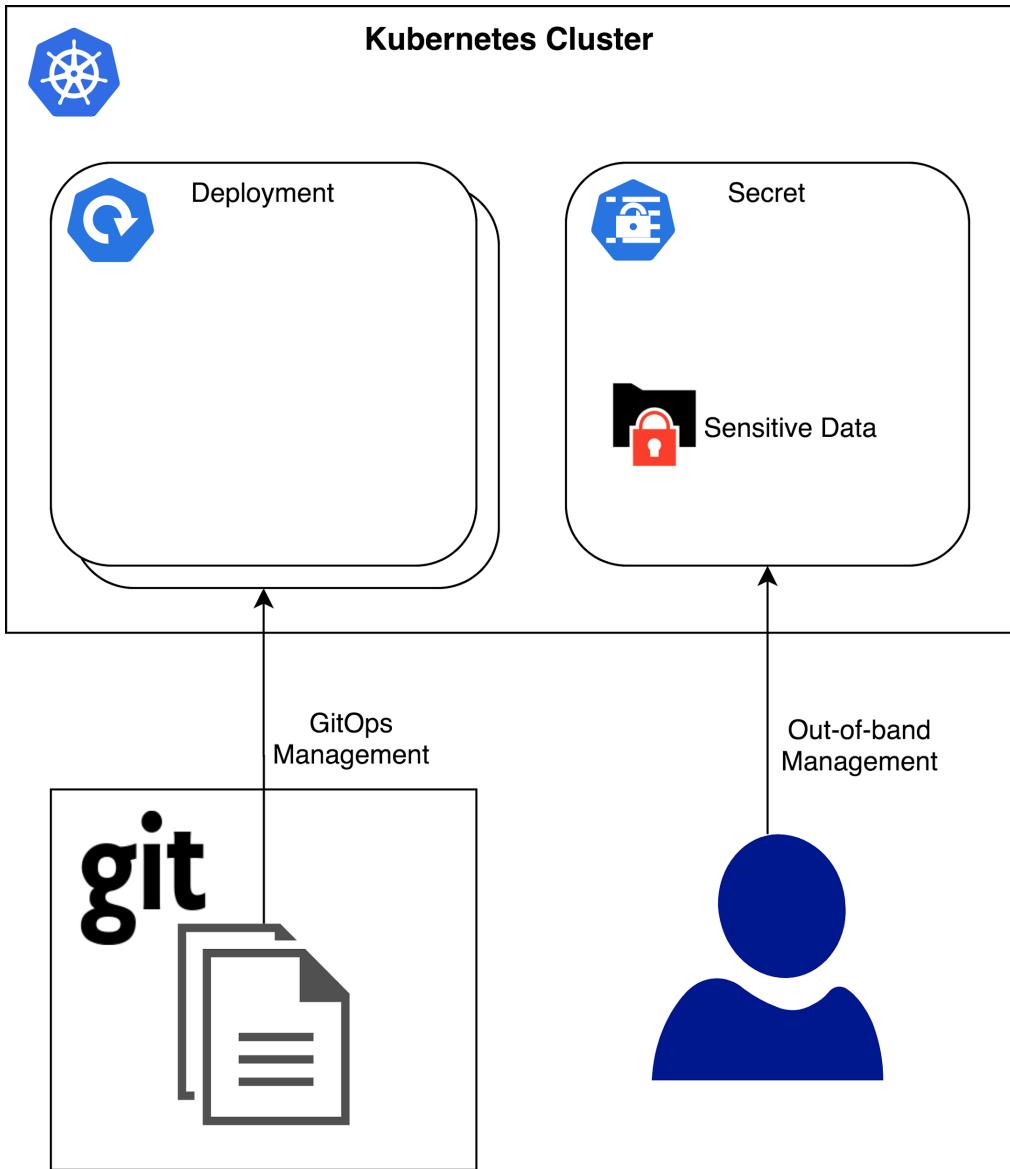


Figure 7.3 Out-of-band Secret management - With out-of-band management, GitOps is used to deploy normal resources. But some other mechanism (e.g. manual kubectl apply) is used to deploy the Secret.

The obvious disadvantage of this approach is that you would need to have two different mechanisms for deploying resources to the cluster. One for normal Kubernetes resources via GitOps, and another strictly for Secrets.

7.3.4 External Secret Management Systems

Another strategy for dealing with secrets in GitOps, is to use an external secret management system *other than* Kubernetes. In this strategy, rather than using the native Kubernetes features to store and load secrets into the container, the application containers themselves retrieve the secret values dynamically at runtime, at the point of use.

A variety of secret management systems exist, but the most popular and widely used one is Hashicorp Vault, which is the tool that we will primarily focus on when discussing external secret management systems. Individual cloud providers also provide their own secret management services such as AWS Secrets Manager, Google Cloud Secret Manager, and Microsoft Azure Key Vault. The tools may differ in capabilities and featuresets, but the general principles are the same, and should be applicable to all.

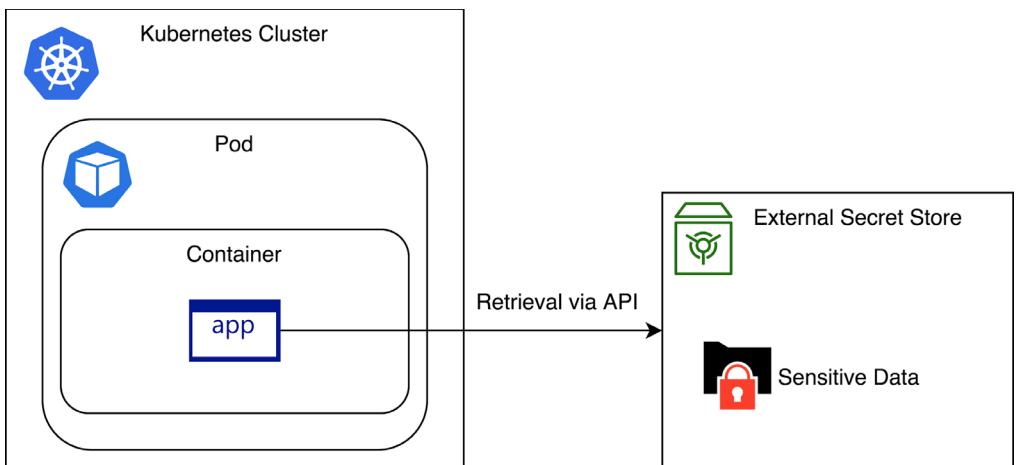


Figure 7.4 Retrieval of secret from external secret store - In this approach, sensitive data is not stored as Kubernetes Secrets. Instead, they are stored on an external system, which would be retrieved by a container at runtime (e.g. via API call).

By choosing to use an external secret management system (such as Vault) to manage your secrets, you are also effectively making a decision **not to use** Kubernetes Secrets. This is because when using this strategy, you are relying on the external secret management system to store and retrieve your secrets, and not Kubernetes. A large consequence of this, is that you also would not be able to leverage some of the conveniences that Kubernetes Secrets provides, such as setting the value of an environment variable from a Secret, or mapping the secret as files in a volume.

When using an external secret store, it is the responsibility of the application to retrieve the secrets from the store securely. For example, when the application starts, it could dynamically retrieve the secret values from the secret store at runtime, as opposed to using the Kubernetes mechanisms (e.g. environment variables, volume mounts). This shifts the

burden of safe-keeping of secrets to both the application developers who must retrieve the secret safely, as well as administrators of the external secret store.

Another consequence of this technique is that because Secrets are managed in a separate database, you do not have the same history/record of when Secrets were changed as you do for your configuration managed in git. This could even affect your ability to rollback in a predictable manner. For example, during a rollback, applying the manifests at the previous git commit might not be enough. You would additionally have to rollback the Secret to previous value at the same time of the git commit. Depending on what secret store was used, this may be inconvenient in the best case, or downright impossible in the worst.

7.3.5 Encrypting Secrets in Git

Since git is considered unsafe for storing plain-text secrets, one strategy is to encrypt the sensitive data so that it *is* safe to store in git, and then decrypt the encrypted data closer to its point of use. The actor performing the decryption would have to have the necessary keys to decrypt the encrypted secret. This might be the application itself, an init container which populates a volume used by the application, or a controller to handle these tasks for the application seamlessly.

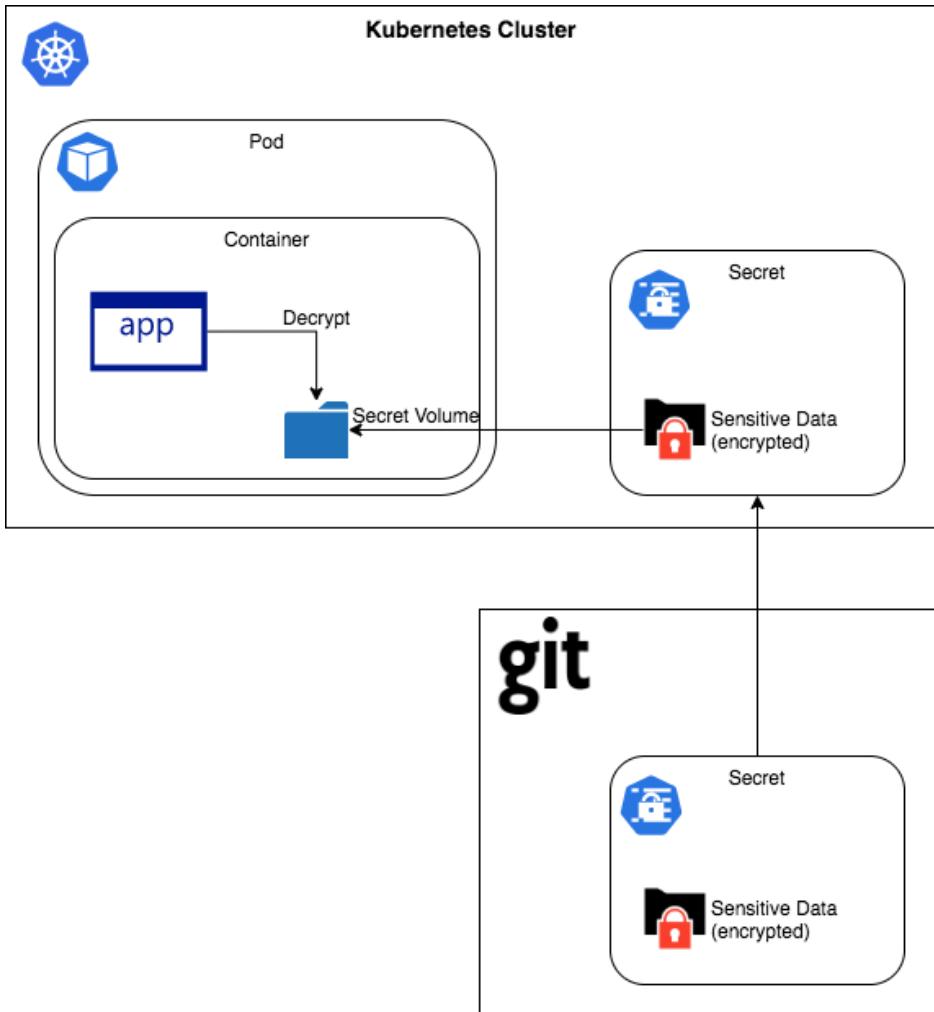


Figure 7.5 Store encrypted secret in Git - Secrets are encrypted and safely stored in git, alongside other Kubernetes resources. At runtime, the application can decrypt the contents before using.

One popular tool which aids in this technique of encrypting secrets in git is Bitnami SealedSecrets, which we will cover in-depth later in this chapter.

The challenges to encrypting secrets in git, is that there still is one last secret involved, and that is the encryption key used to encrypt those secrets. Without adequate protection of the encryption key this technique is meaningless, as anyone with access to the encryption key now has the ability to decrypt and gain access to the sensitive data in the manifests.

7.3.6 Comparison of Strategies

There are many different approaches to managing Secrets in Kubernetes, each with their own tradeoffs. Consider the following advantages and disadvantages before deciding on a solution and/or tool that fits your needs.

Table 2. GitOps Secret Management Strategies

Type	Advantages	Disadvantages
Store in Git	Simple and convenient. Secrets and configuration are managed in the same place (git).	Completely insecure.
Bake Into Image	Simple and convenient.	Container images are sensitive. Rotating secrets requires rebuilding. Images are not portable. Secrets not shareable across pods
Out-of-band Management	Still able to leverage native Kubernetes Secrets faculties (e.g. volumeMounts, environment variables)	Different processes for deploying secrets vs. config. Modifications to secrets are not recorded in git history, possibly affecting the ability to rollback.
External Management System	Most flexible. Can leverage secret management systems other than Kubernetes.	Different process for managing secrets vs. config. API access to the external secret management system still needs to be secured. Loses native K8s faculties for using Secrets (e.g. volumeMounts, environment variables) Modifications to secrets are not recorded in git history, possibly affecting the ability to rollback.
Encrypt in Git	Secrets and configuration are managed in the same place (git). Changes to Secrets are recorded in git	Encryption key needs to be securely managed, which becomes burdensome. Loss of encryption key compromises all

	history. Rollback is predictable and consistent.	secrets.
--	---	----------

7.4 Tooling

Both within and outside the Kubernetes ecosystem, numerous projects have emerged to help users deal with the problem of Secrets. All of the projects use one of the strategies to secret management discussed above. In this section, we cover some of the more popular tools that can complement a Kubernetes environment with a GitOps focused approach.

7.4.1 Hashicorp Vault

Vault, by Hashicorp, is a purpose built, open-source tool for storing and managing secrets in a secure manner. Vault provides a CLI, UI, as well as an API for programmatic access to the secret data. Vault is not specific to Kubernetes, and is popular as a stand-alone secret management system.

Vault Installation and Setup

There are many ways to install and run Vault. But if you are new to Vault, the recommended and easiest way to get started is to install Vault using the official Helm chart maintained by Hashicorp. For the purposes of simplifying our tutorial, we will be installing vault in “dev” mode, which is meant for experimentation, development, and testing. Additionally, the command also installs the Vault Agent Sidecar Injector, which we will cover and use in the following section.

```
# NOTE: requires Helm v3.0+
$ helm repo add hashicorp https://helm.releases.hashicorp.com
$ helm install vault hashicorp/vault \
    --set server.dev.enabled=true \
    --set injector.enabled=true
```

NOTE Non-Kubernetes Installation Note that it is not necessary to run Vault in a Kubernetes environment. Vault is a general purpose secret management system, useful for applications and platforms other than Kubernetes. Many enterprises choose to run a centrally managed Vault instance for their company, so a single Vault instance can service multiple Kubernetes clusters, virtual machines, as well as be accessed by developers and operators from the corporate network and workstations.

The vault CLI can be downloaded from <https://www.vaultproject.io/downloads>, or if on macOS, using the brew package manager.

```
$ brew install vault
```

Once installed, vault can be accessed through standard port-forwarding, and visiting the UI at <http://localhost:8200>.

```
# Run the following from a different terminal, or background it with '&'
```

```
$ kubectl port-forward vault-0 8200
$ export VAULT_ADDR=http://localhost:8200
# In dev mode, the token is the word: root
$ vault login
Token (will be hidden):
Success! You are now authenticated. The token information displayed below
is already stored in the token helper. You do NOT need to run "vault login"
again. Future Vault requests will automatically use this token.

Key          Value
---          -----
token        root
token_accessor o4SQvGgg4ywEv0wnGgqHhK1h
token_duration ``
token_renewable false
token_policies ["root"]
identity_policies []
policies      ["root"]

$ vault status
Key          Value
---          -----
Seal Type    shamir
Initialized   true
Sealed       false
Total Shares 1
Threshold    1
Version      1.4.0
Cluster Name vault-cluster-23e9c708
Cluster ID   543c058a-a9d4-e838-e270-33f7e93814f2
HA Enabled   false
```

Vault Usage

Once Vault is installed in your cluster it's time to store your first secret in vault.

```
$ vault kv put secret/hello foo=world
Key          Value
---          -----
created_time  2020-05-15T12:36:21.956834623Z
deletion_time n/a
destroyed     false
version       1
```

To retrieve the secret, run the `vault kv get` command:

```
$ vault kv get secret/hello
===== Metadata =====
Key          Value
---          -----
created_time  2020-05-15T12:36:21.956834623Z
deletion_time n/a
destroyed     false
version       1

==== Data ===
Key      Value
```

```
--- -----
foo    world
```

By default, `vault kv get` will print the secrets in a tabular format. While this format is presented in an easy to read way and is great for humans, it's not as easy to parse via automation and to be consumed by an application. To aid in this, vault provides some additional ways of formatting the output, and extracting specific fields of the secret:

```
$ vault kv get -field foo secret/hello
world

$ vault kv get -format json secret/hello
{
  "request_id": "825d85e4-8e8b-eab0-6afb-f6c63856b82c",
  "lease_id": "",
  "lease_duration": 0,
  "renewable": false,
  "data": {
    "data": {
      "foo": "world"
    },
    "metadata": {
      "created_time": "2020-05-15T12:36:21.956834623Z",
      "deletion_time": "",
      "destroyed": false,
      "version": 1
    }
  },
  "warnings": null
}
```

This makes it easy for the vault CLI to be used in a startup script which might:

1. Run `vault kv get` command to retrieve the value of a secret.
2. Set the secret value as an environment variable or file.
3. Start the main application which can now read the secret from env var or file.

An example of such a startup script might look like the following:

Listing 7.6 vault-startup.sh.

```
#!/bin/sh

export VAULT_TOKEN=your-vault-token
export VAULT_ADDR=https://your-vault-address.com:8200
export HELLO_SECRET=$(vault kv get -field foo secret/hello)
./guestbook
```

In order to integrate this with a Kubernetes application, the above startup script would be used as the entrypoint to the container, replacing the normal application command with the startup script which starts the application *after* the secret has been retrieved and set to an environment variable.

One thing to notice about this approach, is that the `vault kv get` command *itself* needs privileges to access vault. So for this script to even work, `vault kv get` needs to securely communicate with the Vault server, typically using a vault token. Another way of saying this,

is that you still need a secret to get more secrets. This presents a chicken and egg problem, where you now need to somehow securely configure and store the vault secret needed to retrieve the application secrets. The solution to this lies in a Kubernetes-Vault integration which we will cover in the next section.

7.4.2 Vault Agent Sidecar Injector

Due to its popularity, many Vault and Kubernetes integrations have been created to make it easier to use Vault with Kubernetes. The official Kubernetes integration, developed and supported by Hashicorp, is the Vault Agent Sidecar Injector.

As explained in the previous section regarding Vault, in order for an application to retrieve secrets from Vault, a specialized script was used which performed some prerequisite steps before launching the application. This involved retrieving and preparing the application secrets. There were a few problems with that approach:

1. Although application secret(s) were retrieved in a secure manner, the technique still needed to deal with protecting the vault secret used to access the application secrets.
2. The container needed to be “vault aware,” in the sense that the container needed to be built with a specialized script which understood how to retrieve a specific vault secret and pass it to the application.

To solve this problem, Hashicorp developed the Vault Agent Sidecar Injector, which solves these two problems in a generic way. The Vault Agent Sidecar Injector automatically modifies Pods which are annotated a specific way, and securely retrieves annotated secret references (i.e. application secrets) and renders those values into a shared volume accessible to the application container. By rendering secrets to a shared volume, containers within the pod can consume Vault secrets without being Vault aware.

How It Works

The Vault Agent Injector alters pod specifications to include Vault Agent containers that populate Vault secrets to a shared memory volume accessible to the application. In order to achieve this leverages a feature in Kubernetes called mutating admission webhook.

Mutating admission webhooks: Mutating admission webhooks are one of the many ways to extend Kubernetes API server with additional functionality. Mutating webhooks are implemented as HTTP callbacks, which intercept admission requests (i.e. create, update, patch requests), and modify the object in some way.

The following diagram explains how the Vault Agent Injector works:

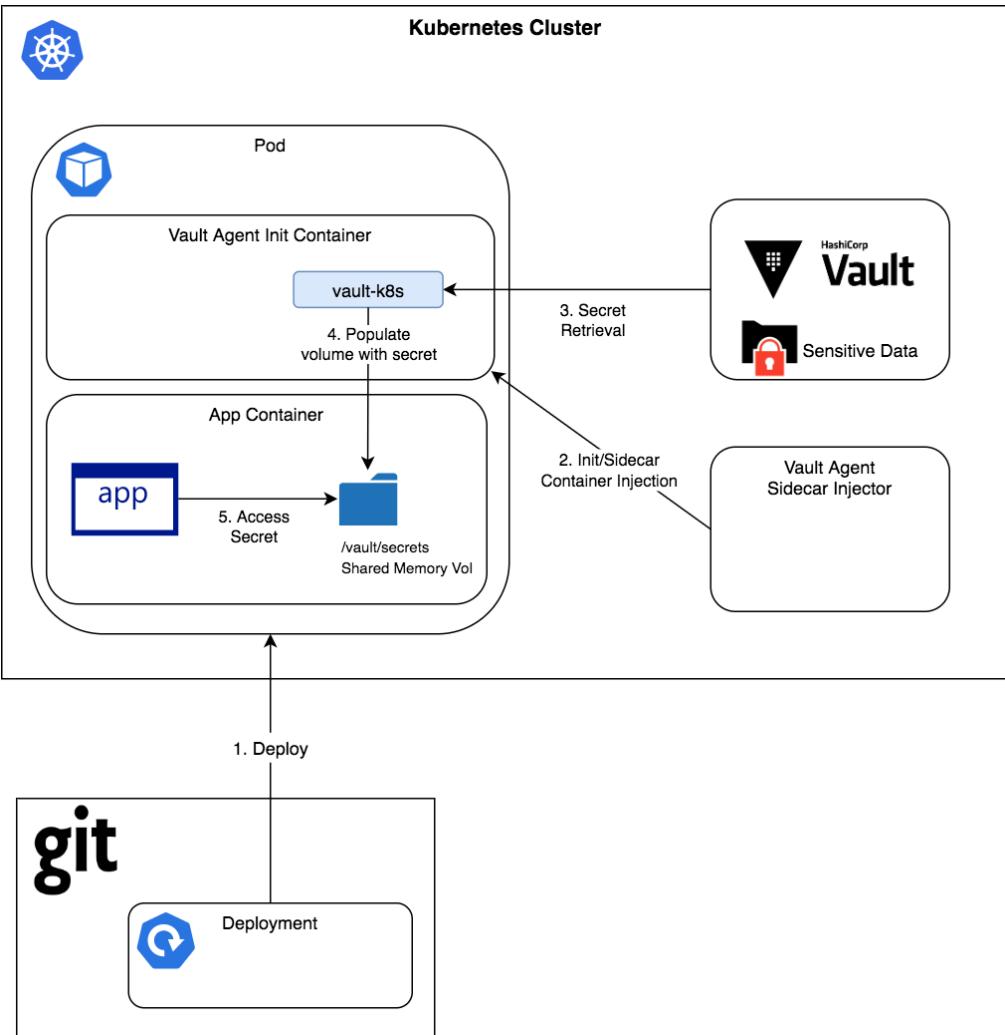


Figure 7.6 Vault Agent Sidecar Injector - A Pod is created normally, but has special annotations which are understood by the Vault Agent Sidecar Injector. Based on the annotations, a directory containing the desired secrets will be mounted into the container for use by the application.

The series of steps involved in this approach are:

1. A workload resource (e.g. Deployment, Job, ReplicaSet, etc..) is deployed to the cluster. This eventually creates a Kubernetes Pod.
2. As the Pod is being created, the Kubernetes API server invokes a mutating webhook call to the Vault Agent Sidecar Injector. The Vault Agent Sidecar Injector modifies the pod by injecting an init container to the pod (and optionally a sidecar).

3. When the Vault Agent Init Container runs, it securely communicates with Vault to retrieve the secret.
4. The secret is written to a shared memory volume, which is shared between the init container and the application container.
5. When the application container runs, it is now able to retrieve the secret from the shared memory volume.

Vault Agent Sidecar Injector Installation and Setup

Earlier in the chapter, we describe how to install Vault using the official Helm chart. This chart also includes the agent sidecar injector. The instructions are repeated here. Note that the examples assume your current kubectl context is pointing at the “default” namespace.

```
# NOTE: requires Helm v3.0+
$ helm repo add hashicorp https://helm.releases.hashicorp.com
$ helm install vault hashicorp/vault \
  --set server.dev.enabled=true \
  --set injector.enabled=true
```

Usage

When an application desires to retrieve its secrets from Vault, the pod spec needs to have at a minimum, the following the vault agent annotations:

Listing 7.7 vault-agent-inject-annotations.yaml.

```
annotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/agent-inject-secret-hello.txt: secret/hello
  vault.hashicorp.com/role: app
```

Breaking this down, the above annotations conveys a couple of pieces of information:

1. The annotation key `vault.hashicorp.com/agent-inject: "true"`, informs Vault Agent Sidecar Injector that Vault secret injection should occur for this pod.
2. The annotation value `secret/hello`, indicates which Vault secret key to inject into the pod.
3. The suffix `.txt` of the annotation, `vault.hashicorp.com/agent-inject-secret-hello.txt` indicates that the secret should be populated under a file named `hello.txt` in the shared memory volume with the final path being `/vault/secrets/hello.txt`.
4. The annotation value from the `vault.hashicorp.com/role` indicates which Vault role should be used when retrieving the secret.

Now let’s try with a real example. To run all the vault commands in this tutorial, you will need to first gain console access inside the vault. Run `kubectl exec`, to access the interactive console of the vault server:

```
$ kubectl exec -it vault-0 -- /bin/sh
/ $
```

If you haven't already, follow the earlier guide on creating your first secret, named "hello" in vault:

```
$ vault kv put secret/hello foo=world
Key          Value
---          -----
created_time 2020-05-15T12:36:21.956834623Z
deletion_time n/a
destroyed     false
version       1
```

Next we need to configure vault to allow kubernetes pods to authenticate and retrieve secrets. To do so, run the follow vault commands to enable the kubernetes auth method:

```
$ vault auth enable kubernetes
Success! Enabled kubernetes auth method at: kubernetes/

$ vault write auth/kubernetes/config \
  token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
  kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \
  kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
Success! Data written to: auth/kubernetes/config
```

The above two commands configure vault to use the Kubernetes authentication method to use the service account token, the location of the Kubernetes host, and its certificate.

Next, we define a policy named "app", as well as a role named "app", which will have read privileges to the "hello" secret.

```
# Create a policy "app" which will have read privileges to the "secret/hello" secret
$ vault policy write app - <<EOF
path "secret/hello" {
  capabilities = ["read"]
}
EOF

# Grants a pod in the "default" namespace using the "default" service account
# privileges to read the "hello" secret
$ vault write auth/kubernetes/role/app \
  bound_service_account_names=default \
  bound_service_account_namespaces=default \
  policies=app \
  ttl=24h
```

Now it's time to deploy a pod which will automatically get our injected Vault secret. Apply the following Deployment manifest which has the vault annotations we described earlier on the Pod:

Listing 7.8 vault-agent-inject-example.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vault-agent-inject-example
spec:
  selector:
    matchLabels:
```

```

    app: vault-agent-inject-example
template:
  metadata:
    labels:
      app: vault-agent-inject-example
    annotations:
      vault.hashicorp.com/agent-inject: "true"
      vault.hashicorp.com/agent-inject-secret-hello.txt: secret/hello
      vault.hashicorp.com/role: app
spec:
  containers:
    - name: debian
      image: debian:latest
      command: [sleep, infinity]

```

When the deployment is up and running, we can access the console of the pod and verify that the pod does indeed have the secret mounted into the Pod:

```

$ kubectl exec deploy/vault-agent-inject-example -it -c debian - bash
root@vault-agent-inject-example-5c48967c97-hgzds:/# cat /vault/secrets/hello.txt
data: map[foo:world]
metadata: map[created_time:2020-10-14T17:58:34.5584858Z deletion_time: destroyed:false
version:1]

```

Also you can see, using the Vault Agent Sidecar Injector is one of the easiest ways to get Vault secrets seamlessly into your pods in a secure manner.

7.4.3 Sealed Secrets

SealedSecrets, by Bitnami, is another solution to the GitOps secret problem, and aptly describes the problem as: "I can manage all my K8s config in git, except Secrets." While not the only tool, currently SealedSecrets is the most popular and widely used tool for teams who would prefer to encrypt their Secrets in git. This allows everything, including secrets to be completely and wholly managed in git.

Sealed Secrets follows the strategy of encrypting the sensitive data so that it can be safely stored in git, and decrypting it inside the cluster. What makes it unique, is that it provides a controller and command line interface, which helps automate this process.

How It Works

Sealed Secrets is comprised of:

- A new CustomResourceDefinition, called a SealedSecret, which will produce a normal Secret.
- A controller which runs in the cluster which is responsible for decrypting a SealedSecret, and producing a normal Kubernetes Secret with the decrypted data.
- A command line tool, kubeseal, which encrypts sensitive data into a SealedSecret for safe storage in git.

When a user wishes to manage a secret using git, they will "seal" or encrypt the secret into a SealedSecret custom resource using the kubeseal CLI, which they would store in git,

alongside other application resources (e.g. Deployments, ConfigMaps, etc...). The SealedSecret is deployed like any other kubernetes resource.

When a SealedSecret is deployed, the sealed-secrets-controller will decrypt the data and produce a normal kubernetes Secret with the same name. At this point, there is no difference in the experience from a SealedSecret and normal kubernetes Secrets, since a regular kubernetes Secret is available to be used by Pods.

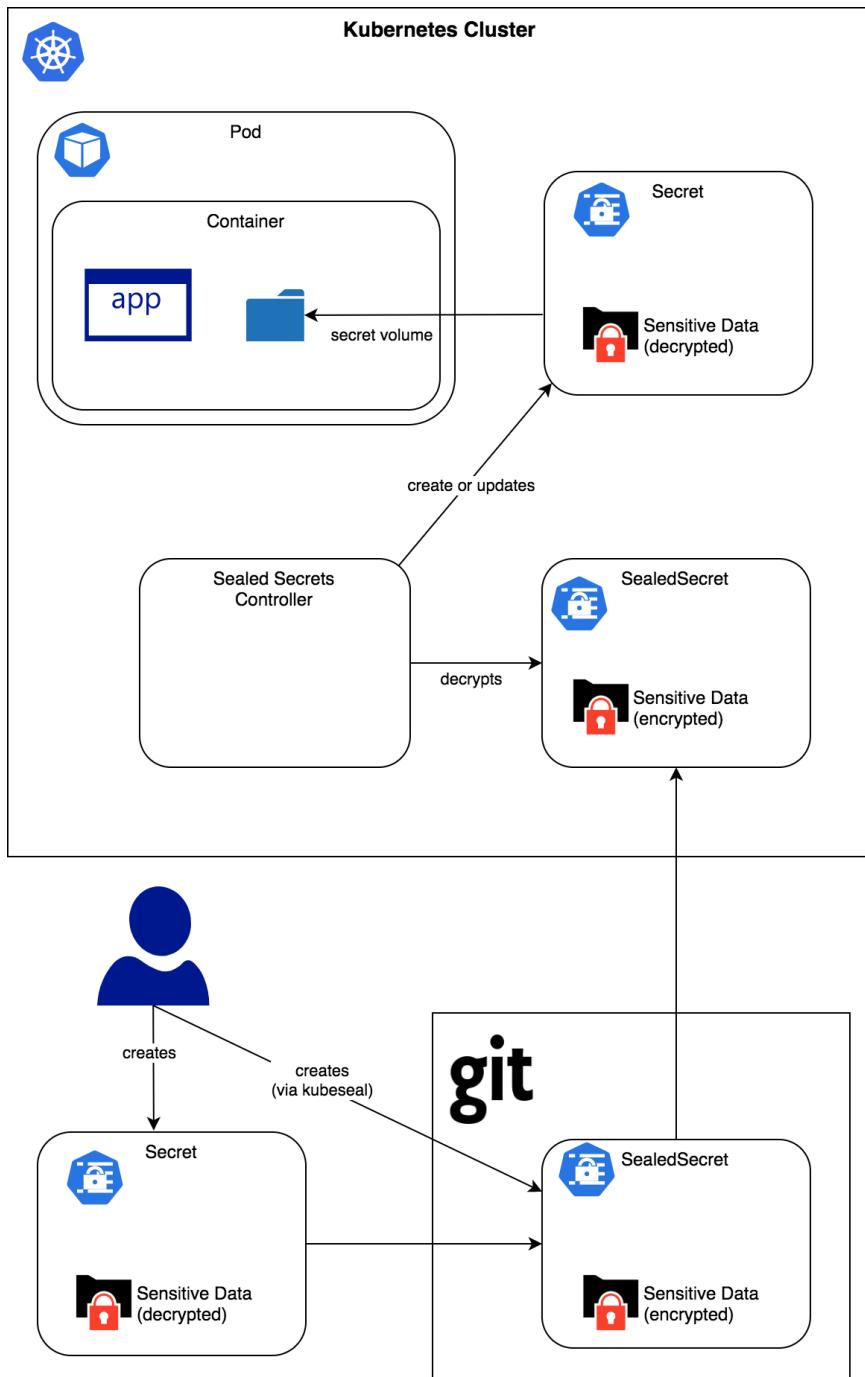


Figure 7.7 Sealed Secrets - A user will encrypt a Secret into a SealedSecret to store in git. The Sealed Secrets controller decrypts the SealedSecret and formulates a corresponding Kubernetes Secret to be used by a Pod using normal Kubernetes faculties.

INSTALLATION

CRD and Controller:

```
$ kubectl apply -f https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.12.4/controller.yaml
```

Kubeseal CLI:

- Download binary from <https://github.com/bitnami-labs/sealed-secrets/releases>

USAGE

To use SealedSecrets, you first create a regular kubernetes Secret like you normally would, using your preferred technique, and place it at some local file path. For this simple example, we will use the `kubectl create secret` command to create a password secret. The `--dry-run` flag is used to print the value to stdout, which is then redirected to a temporary file. We store it in a temporary location, since the secret containing the unencrypted data should be discarded and not persisted in git (or anywhere else).

```
$ kubectl create secret generic my-password --from-literal=password=Pa55Word1 --dry-run -o yaml > /tmp/my-password.yaml
```

NOTE *Don't use kubectl create secret --from-literal* The use of `--from-literal` in the above example, is only for demo and exercise purposes, and should never be used with any sensitive data. This is because your shell records recently run commands into a history file for convenient retrieval. If you wish to use `kubectl` to generate a secret, consider using `--from-file` instead.

The above command will produce the following temporary Kubernetes secret file:

Listing 7.9 my-password.yaml.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
data:
  password: UGE1NVdvcmQx
```

The next step is to “seal” or encrypt the temporary secret using the `kubeseal` CLI. The following command creates a SealedSecret object, based on the temporary secret file that we just created.

```
$ kubeseal -o yaml </tmp/my-password.yaml > my-sealed-password.yaml
```

This will produce the following SealedSecret resource which can be safely stored in git.

Listing 7.10 my-sealed-password.yaml.

```

apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: my-password
  namespace: default
spec:
  encryptedData:
    password:
      AgAF7r6v4LG/JPU7Ti0i77bhd1NJ9ua9gldvzNw7wBKK2LLJyndSR8GShF3f1zRY+cNM0i0GTkcaFrNRCG/C
      MrLiwNltQv1gZKqryFugjcp7tiM0dwmmi4M0aIeqRfxq3+vL/Mmdc/xEsK/FtuK0g18rWoG/wEhvNhtvXu1t
      4kXHTSVL5xa4KmYD8Hn8p8CNZrGATLfy6rI1ZsydM9DoB1nSFdsfG5kH1E++RbyXxd6Y6vcck1DP16oq15Gi
      dnrEQ1QmkhEr+h/YuUrajAxMFNZpqzs9yaTkURdc0xP2wMiycBooEn7eRzTt2aToh04q9rgoiWwjztCyx0
      CyCt+eisoG0QsqC697PiQV35IFuNbKptyFUU04nfMtXyfb2aZEHFvt8/j3x19JlqKQ16zy9g0jhj10LxhBjm
      RK9EyqTxqVGRTfrHaHqqz7mzSy/x2H61kfBBVFLSwv0FkYD82wdQRFTYBF5Uu/cnjeB2Uob8JkM91nEtXhL
      WAwtl2K5w0LYyUd3q0aNEEXgyv+dN/4pTHK1V+LF6IHND0Fau8QVNmqJrxrXv8yEnRGzBYg60J99K19vhph8p
      fbHAYFn2Tb9o8WxkjWD0YAc+pAuFAGjUmJKEJmaPr0vUo0k67B1Xj77LVuHPH6Ei6JxGYOZA0B2WE1m0wILH
      zD17unWXnI+Q7Hmk2TEYSeEo81x+I9mLd8D6Epung21Fn=-
template:
  metadata:
    creationTimestamp: null
    name: my-password
    namespace: default

```

The SealedSecret can now be stored alongside your other application manifests, and deployed like a normal resource without any special treatment.

NOTE *Sealing Secrets without cluster access* By default, kubeseal will encrypt the secret using the certificate of the sealed-secrets-controller. To do so, it needs access to the kubernetes cluster to retrieve the certificate directly from the sealed-secrets-controller. It is possible to seal secrets offline without direct access to the cluster. Alternatively, the certificate could be provided using different means, by using the `kubeseal --cert` flag which allows you to specify a local path to a certificate, or even URL.

One thing you might have noticed when comparing the original kubernetes Secret and the SealedSecret, is that the SealedSecret has a namespace specified in the metadata, whereas the original Secret did not. This means that the SealedSecret is much less portable than the original Secret, since it cannot be deployed in different namespaces. This is actually a security feature of SealedSecrets, in what is referred to as a “strict scope.” With a strict scope, SealedSecrets encrypts the Secret in such a way that it can only be used for the namespace that it was encrypted for, and also with the exact same Secret name (my-password in our example). This feature prevents an attack where an attacker could deploy a SealedSecret to a different namespace or name that the attacker has privileges to, in order to view the sensitive data of the resulting Secret.

In non multi-tenant environments, the “strict” scope can be relaxed so that the same SealedSecret can be used in any namespace in the cluster. To do so, you can specify the `--scope cluster-wide` flag during the sealing process.

```
$ kubeseal -o yaml --scope cluster-wide </tmp/my-password.yaml > my-cluster-sealed-
password.yaml
```

This produces a slightly different “cluster-scope” SealedSecret, which now no longer contains the namespace.

Listing 7.11 my-clusterwide-sealed-password.yaml.

```
apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  annotations:
    sealedsecrets.bitnami.com/cluster-wide: "true"
  creationTimestamp: null
  name: my-password
spec:
  encryptedData:
    password:
      AgBjcpeaU2SKq0TMQ2SxYnxoJgy19PR7uzi1qrP5e3PqCPrI7yWD6TvozJE2r90rey0zLL0/yTuIHn0Z5S7F
      BQT6p7FA19FGxcCu+Xdd1p/purofibL5xR8Zfk/VxEAH2RSVPSUGdMwpMRqhKFrsk2rZujjrDj0dc/7zTRgu
      eSMJ6RTIWSCTxz5htaWIBvN3nUJKGAWsrG/cF1xA6pPANE6eZTjyX3+pEQ3YPmPqkc4chseU/aUqk3fxN5t
      EcwuLWFxFKihN5hMnhKUH8CePk7IWB/BXATxLNY1GRzrcYAoXZ0yYGkUlw24yVM10AbpmlmqYiCdlNQMEhTi
      lc9iyKKT3ASplH+T/WMr7DdkCdpbTcgL0wI0EeBtUXVzBwdnWlquVA6oPCJmo4TruiBtLDZjeu6xj9fVt1ZD
      /HETGLgeDuBSw/BN7fUqi6GuR0bFMiZUhoN4ynm2jNHTe0bVDV6Q0idbTvy6FcPjHVqjwKslu2jN/TYhLTkb
      zHjL90r2dZX8gI/BrmM0toRDzSK2C4T9KqyAxipRgYkSH9cImdT9ChCPA9jIQUZRZGMS48Yg/SDRvA/d+QaG
      dMhhbhtmApwPWMaA/0+adxnPcoKBnvTuzAlPlaYN64JCbzyJkKDvutm/wvMYtoZ95vMnLDG1d/b9CmYobAye
      uz9AGZ5UeZWoZ32DMMhc5kXecR/FsnfMWeCaHiT+6423nJu
  template:
    metadata:
      annotations:
        sealedsecrets.bitnami.com/cluster-wide: "true"
      creationTimestamp: null
      name: my-password
```

One of the consequences of a “cluster-wide” SealedSecret scope, is that the SealedSecret can now be deployed and decrypted in *any* namespace of the cluster. This means that anyone with privileges to a single namespace in the cluster, can simply deploy the SealedSecret in their namespace, for the purposes of viewing the sensitive data.

One challenge to using SealedSecrets, is that the encryption key which is used to encrypt the Secrets, is different for every cluster. When a sealed-secrets-controller converts and encrypts a normal Kubernetes Secrets into a SealedSecrets, that SealedSecret object is only valid for the signing controller, and nowhere else. This means that there is a different SealedSecret object in git for every cluster. If dealing with a single cluster, this challenge may not be an issue for you. However, if the same Secret needs to be deployed to more clusters, then this becomes a configuration management problem, since the SealedSecret would need to be produced for each environment.

Although it is possible to use the same encryption key for multiple clusters, this presents a different challenge: it then becomes difficult to safely distribute, manage, and secure that key in all clusters. The encryption key would be distributed across many locations, presenting more opportunities for it to become compromised, and ultimately allowing the attacker to gain access to every Secret in every cluster.

7.4.4 Kustomize Secret Generator Plugin

Users who are managing their kubernetes configurations using Kustomize have a unique feature available to them, Kustomize plugins, which can be leveraged for retrieving secrets. Kustomize's Plugin feature allows kustomize to invoke user defined logic to generate or transform kubernetes resources during the build process. Plugins are very powerful, and could be written to retrieve secrets from an external source, such as a database, RPC calls, or even an external secret store such as Vault. The plugin could even be written to perform decryption of encrypted data and transform it into a Kubernetes secret. The important takeaway is that Kustomize plugins provide a very flexible mechanism for producing Secrets, and can be implemented with any logic which suits your needs.

How It Works

As we learned in previous chapters, Kustomize is a configuration management tool, and is not normally in the business of managing or retrieving secrets. But by using the plugin functionality of Kustomize, it is possible to inject some user defined logic to generate kubernetes manifests as part of a `kustomize build` command. Since `kustomize build` is often the last step to occur before the actual deployment of the rendered manifests, it is a perfect opportunity to perform a secure retrieval of a secret before it is deployed, and ultimately avoid the storage of secrets in git.

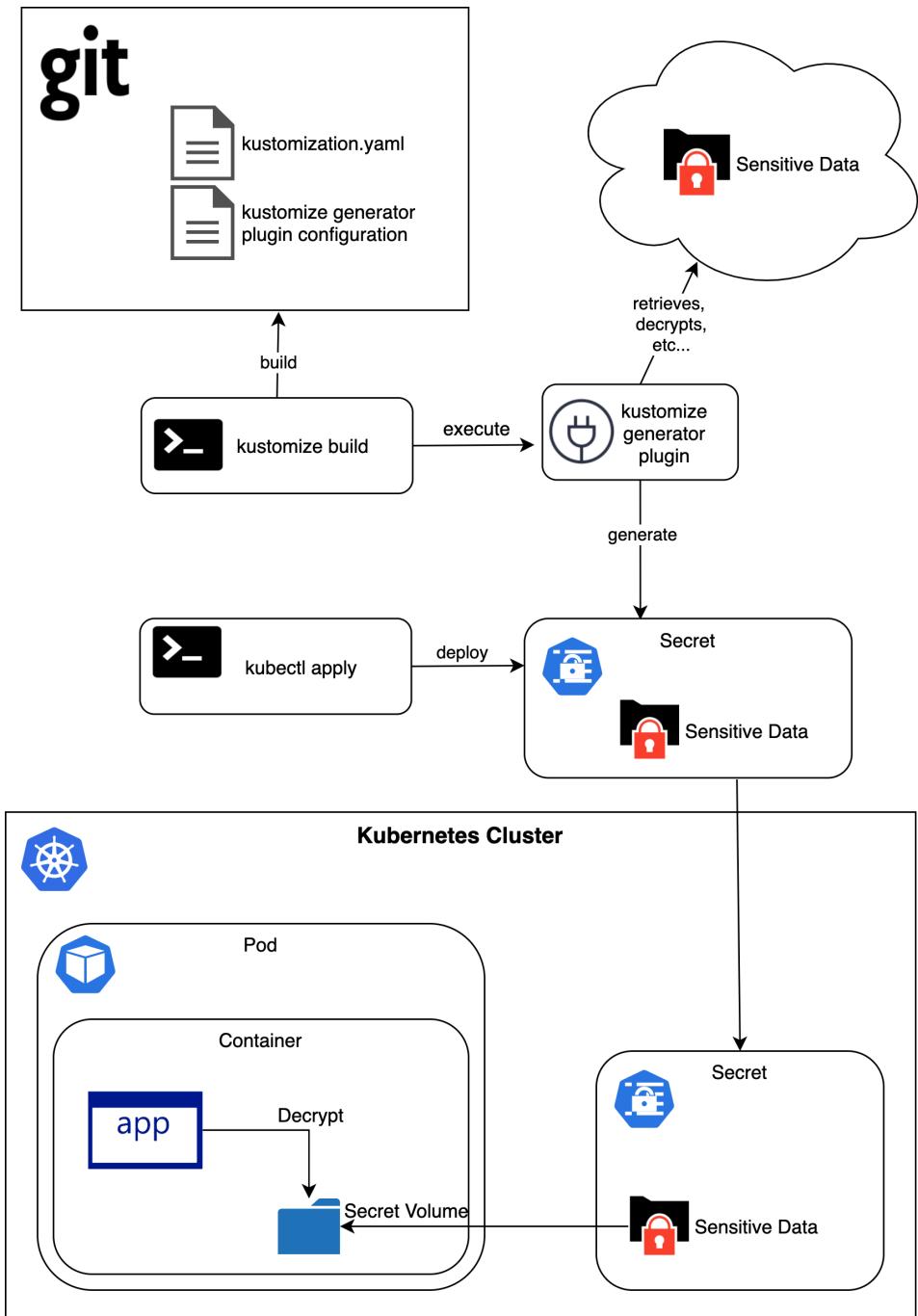


Figure 7.8 Kustomize Secret Generator Plugin - Instead of storing Secrets in git, the recipe for formulating or retrieving that Secret is stored (as a Kustomize secret generator). This approach implies the rendered Secret will be applied to the cluster immediately after rendering (the kustomize build).

Kustomize has two types of plugins: exec plugins, and Go plugins. Exec plugins are simply an executable which accepts a single command line argument: the path of the plugin YAML configuration file. Also supported are Go plugins, which are written in Golang, but are more complex to develop. In the following exercise, we will be writing an exec plugin, since it is simpler to write and understand.

For this exercise, we will be implementing a Kustomize Secret Retriever plugin, which will “retrieve” a specific secret by a key name, and generate a Kubernetes Secret from it. The word “retrieve” is in quotes, because in reality, this example will simply pretend to retrieve a secret, and use a hardwired value instead.

To use a kustomize generator plugin, we simply reference the plugin configuration in the `kustomization.yaml`.

Listing 7.12 kustomization.yaml.

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

generators:
- my-password.yaml
```

The contents of referenced plugin configuration YAML, is specific to the plugin. There is no standard about what belongs in a Kustomize plugin manifest specification. In this exercise, our plugin specification is very simple, and contains only two pieces of necessary information:

- The name of the Kubernetes secret to create (we will use the same name as the plugin configuration name)
- The key in an external secret store to retrieve (which the plugin will pretend to get)

Listing 7.13 my-password.yaml.

```
apiVersion: gitopsbook #A
kind: SecretRetriever
metadata:
  name: my-password    #B
spec:
  keyName: foo          #C
```

#A the `apiVersion` and `kind` fields are used by kustomize to discover which plugin to run

#B in this example, the plugin will choose to use the configuration name as the resulting K8s Secret name. However kustomize plugins are free to ignore `metadata.name` completely.

#C `keyName` will be the key which will be “retrieved” from the external store

Finally, we get to the actual plugin implementation, which we will be writing as a shell script. This plugin accepts the path of the plugin configuration, and parses out the secret name and key to retrieve to generate a final Kubernetes Secret to deploy.

Listing 7.14 gitopsbook/secretretriever/SecretRetriever.

```
#!/bin/bash

#A
config=$(cat $1)

#B
secretName=$(echo "$config" | grep "name:" | awk -F: '{print $2}')

#C
keyName=$(echo "$config" | grep "keyName:" | awk -F: '{print $2}')

#D
password="Pa55w0rd!"

#E
base64password=$(echo -n $password | base64)

#F
echo "
kind: Secret
apiVersion: v1
metadata:
  name: $secretName
data:
  $keyName: $base64password

#A The first argument to a kustomize plugin, is the path to the plugin configuration file, referenced in the
#      kustomization.yaml. This line simply grabs its contents.
#B Parse the plugin config, and use the name of the config as the name of the resulting K8s Secret
#C Parse the keyName from the plugin config, and use it as a key in the K8s Secret
#D For demo purposes, we use a hard-wired value. This would typically be replaced with logic which retrieves and/or
#      decrypts secrets.
#E Kubernetes secrets need to be base64 encoded
#F Print the final Kubernetes secret to standard out
```

This example can also be run from the GitOps resources git repository.

```
$ git clone https://github.com/gitopsbook/resources
$ cd resources/chapter-07/kustomize-secret-plugin
$ export KUSTOMIZE_PLUGIN_HOME=$(pwd)/plugin
$ kustomize build ./config --enable_alpha_plugins
apiVersion: v1
data:
  foo: UGE1NXcwcmQh
kind: Secret
metadata:
  name: my-password
```

Using kustomize plugins, it's possible to choose virtually any technique to generate a Secret, including all of the different strategies mentioned in this chapter. This includes retrieving a secret by a reference, decrypting an encrypted secret in git, accessing a Secret management

system, etc... The options are left to the user, on which strategy makes the most sense for their situation.

7.5 Summary

- Kubernetes Secrets is a simple data structure which allows the separation of the configuration of an application from the build artifact.
- Kubernetes Secrets can be used by Pods in various ways, including volume mounts, environment variables, or direct retrieval from the Kubernetes API.
- Git is not appropriate for secrets due to lack of encryption and path level access control.
- Baking secrets into the container means the container itself is also sensitive and no separation of configuration from the build artifact.
- Out-of-band secret management allows native Kubernetes faculties to be used, but results in different mechanisms to manage/deploy Secrets vs. config
- External secret management allows flexibility, but loses the ability to use Kubernetes native Secret faculties.
- Hashicorp vault is a secured external secret store and can be installed using brew. Vault also provides a CLI “vault” to manage the secret in the store. Pods on startup can fetch the secrets from the external store using the CLI and scripting.
- Vault agent sidecar injector can automate the injection of secrets into the pods without the CLI and scripting.
- Sealed Secrets is a Custom Resource Definition (CRD) for securing the data in Kubernetes Secrets. Sealed Secrets can be installed to the cluster by applying the Sealed Secrets manifest. Sealed Secrets comes with a CLI tool “kubeseal” to encrypt the data in Kubernetes Secrets.
- Kustomize Secret Generator Plugin enables user defined logic to inject secrets in manifest during the build process.

8

Observability

This chapter covers:

- Relating GitOps to observability
- Providing observability of Kubernetes to a cluster operator
- Enabling GitOps through Kubernetes observability
- Improving the observability of the system with GitOps
- Using tools and techniques to ensure your cloud-native applications are also observable

Observability is vital to manage a system properly, determine if it is working correctly, and decide what changes are needed to fix, change, or improve its behavior (i.e., how to control the system). Observability has been an area of interest in the cloud-native community for some time, with many projects and products being developed to allow observability of systems and applications. The Cloud Native Computing Foundation recently formed a Special Interest Group (SIG) dedicated to Observability.⁵³

In this chapter, we will discuss observability in the context of GitOps and Kubernetes. As was mentioned earlier, GitOps is implemented by a GitOps Operator (or controller) or Service that must manage and control the Kubernetes cluster. The key functionality of GitOps is comparing the desired state of the system (which is stored in Git) to the current actual state of the system and performing the required operations to converge the two. This means GitOps relies on the observability of Kubernetes and the application to do its job. But GitOps is also a system that must provide observability. We will explore both aspects of GitOps observability.

We recommend you read chapters 1 and 2 before reading this chapter.

⁵³ <https://github.com/cncf/sig-observability>

8.1 What is Observability?

Observability is a system's capability, like reliability, scalability, or security, that must be designed and implemented into the system during system design, coding, and testing. In this section, we will explore the various ways GitOps and Kubernetes provide observability for a cluster. For example, what version of an application was most recently deployed to the cluster? Who deployed it? How many replicas were configured for the application a month ago? Can a decrease in application performance be correlated to changes to the application or Kubernetes configuration?

When managing a system, the focus is on controlling that system and applying changes that *improve* the system in some way, whether through additional functionality, increasing performance, improving stability, or some other beneficial change. But how do you know how to control the system and what changes to make? Once the changes are applied, how do you know that they improved the system and didn't make it worse?

Remember back to the earlier chapters. We previously discussed how GitOps stores the desired state of a system in a declarative format in Git. A GitOps operator (or service) changes (controls) the system's running state to match the system's desired state. The GitOps operator must be able to observe the system being managed, which in our case is Kubernetes and the applications running on Kubernetes. What's more, the GitOps operator itself must also provide observability so that ultimately the user can control GitOps.

OK, but in practical terms, what does that really mean?

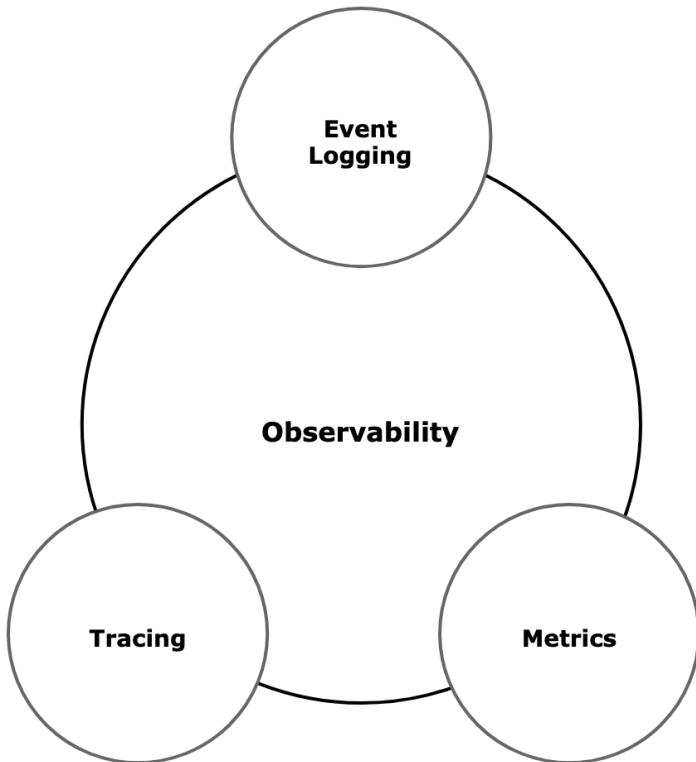


Figure 8.1 Observability is composed of three primary aspects: Event Logging, Metrics, and Tracing. These aspects combine to provide operational insights that enable the proper management of the system.

As was mentioned earlier, Observability is a capability of a system that encompasses multiple aspects. Each of these aspects must be designed and built into the system. Let's briefly examine each of these three aspects: Event Logging, Metrics, and Tracing.

8.1.1 Event Logging

Most developers are familiar with the concept of logging. As the code executes, log messages can be output that indicates significant events, errors, or changes. Each event log is timestamped and is an immutable record of a particular system component's internal operation. When rare or unpredictable failures occur, the event log may provide context at a fine level of granularity, indicating what went wrong.

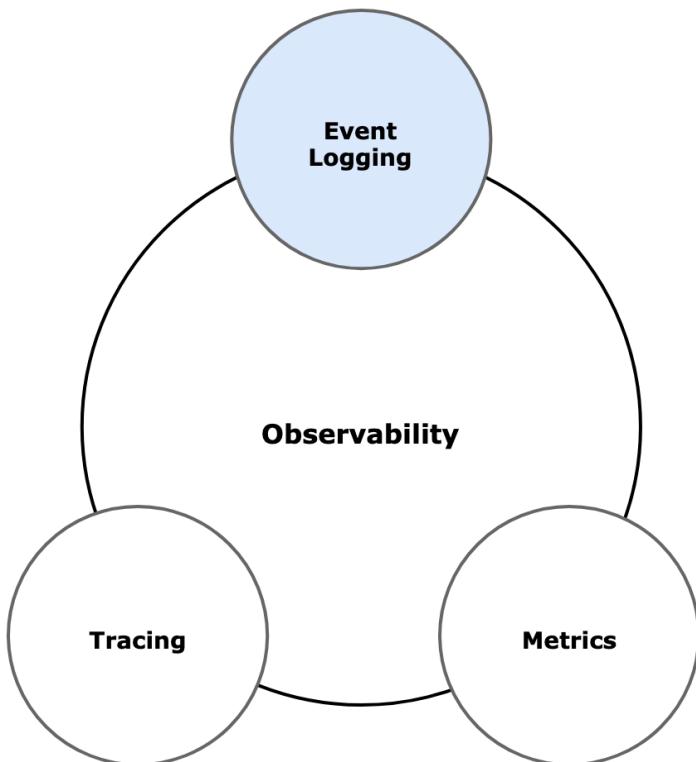


Figure 8.2 Event logs are time-stamped and provides an immutable record of a particular system component's internal operation.

Often the first step in debugging an improperly behaving application is to look at the application's log for clues. Logging is an invaluable tool for developers to observe and debug the system and applications. Logging and the retention of records may also be required for compliance with applicable industry standards.

For Kubernetes, a fundamental aspect of observability is displaying the log output of all the various pods in the cluster. Applications output debug information about their running state to `stdout` ("Standard Out") captured by Kubernetes and saved to a file on the Kubernetes node that the pod/container is running on. The logs of a particular pod can be displayed with the `kubectl logs <pod_name>` command.

To illustrate various aspects of Logging, Metrics, and Tracing, we will be using an example ride-sharing application called Hot ROD.⁵⁴ Let's launch the application in our minikube cluster so we can take a look at its logs. Here is the manifest for the application Deployment.

⁵⁴ <https://github.com/jaegertracing/jaeger/tree/master/examples/hotrod>

Listing 8.1 Hot ROD Application Deployment ([hotrod-app.yaml](#))

```

apiVersion: v1
kind: List
items:
  - apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: hotrod
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: hotrod
      template:
        metadata:
          labels:
            app: hotrod
        spec:
          containers:
            - args: ["-m", "prometheus", "all"]
              image: jaegertracing/example-hotrod:latest
              name: hotrod
              ports:
                - name: frontend
                  containerPort: 8080
  - apiVersion: v1
    kind: Service
    metadata:
      name: hotrod-frontend
    spec:
      type: LoadBalancer
      ports:
        - port: 80
          targetPort: 8080
      selector:
        app: hotrod

```

Deploy the Hot ROD application using the following commands:

```

$ minikube start
$ kubectl apply -f hotrod-app.yaml
deployment.apps/hotrod created
service/hotrod-frontend created

```

Now let's look at the log messages of the Pod.

```

$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
hotrod-59c88cd9c7-dxd55  1/1     Running   0          6m4s

$ kubectl logs hotrod-59c88cd9c7-dxd55
2020-07-10T02:19:56.940Z  INFO      cobra@v0.0.3/command.go:792      Using Prometheus as
                           metrics backend
2020-07-10T02:19:56.940Z  INFO      cobra@v0.0.3/command.go:762      Starting all services
2020-07-10T02:19:57.048Z  INFO      route/server.go:57      Starting {"service": "route",
                           "address": "http://0.0.0.0:8083"}
2020-07-10T02:19:57.049Z  INFO      frontend/server.go:67      Starting {"service":
                           "frontend", "address": "http://0.0.0.0:8080"}

```

```
2020-07-10T02:19:57.153Z INFO customer/server.go:55 Starting {"service": "customer", "address": "http://0.0.0.0:8081"}
```

The output tells us that each of the microservices (route, frontend, and customer) is “Starting.” But at this point, there is not too much information in the log. And it may not be entirely clear if each of the microservices was successfully started.

Use the `minikube service hotrod-frontend` command to create a tunnel on your workstation to the `hotrod-frontend` Service and open the URL in a web browser.

```
$ minikube service hotrod-frontend
|-----|-----|-----|-----|
| NAMESPACE | NAME | TARGET PORT | URL |
|-----|-----|-----|-----|
| default | hotrod-frontend | 80 | http://172.17.0.2:31725 |
|-----|-----|-----|-----|
✖ Starting tunnel for service hotrod-frontend.
|-----|-----|-----|-----|
| NAMESPACE | NAME | TARGET PORT | URL |
|-----|-----|-----|-----|
| default | hotrod-frontend | | http://127.0.0.1:53457 |
|-----|-----|-----|-----|
✖ Opening service default/hotrod-frontend in default browser...
```

This will open a web browser to the application. When it opens, click on each of the buttons to simulate requesting a ride for each customer.

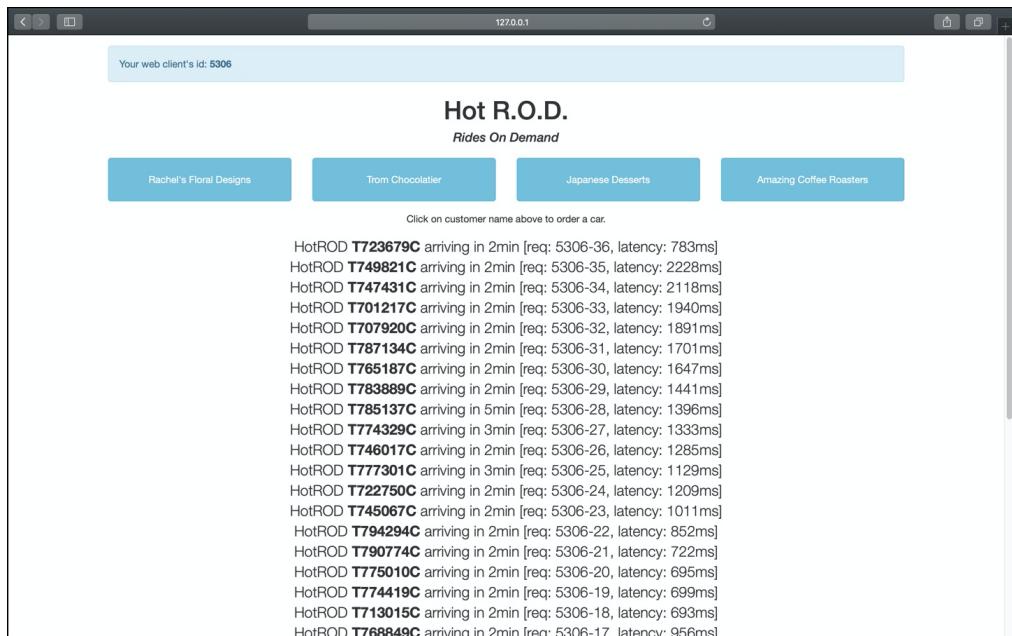


Figure 8.3 Above is a screenshot of the Hot ROD example application that simulates a ride-sharing system. Clicking the buttons at the top of the page initiates a process that invokes multiple microservices to match a

customer with a driver and a route.

Now, in another terminal window, let's look at the logs for the application.

```
$ kubectl logs hotrod-59c88cd9c7-dxd55
:
2020-07-10T03:02:13.012Z INFO frontend/server.go:81      HTTP request received
  {"service": "frontend", "method": "GET", "url": "/dispatch?customer=567&nonce=0.6850439192313089"}
2020-07-10T03:02:13.012Z INFO customer/client.go:54      Getting customer {"service": "frontend", "component": "customer_client", "customer_id": "567"}
http://0.0.0.0:8081/customer?customer=567
2020-07-10T03:02:13.015Z INFO customer/server.go:67      HTTP request received
  {"service": "customer", "method": "GET", "url": "/customer?customer=567"}
2020-07-10T03:02:13.015Z INFO customer/database.go:73      Loading customer {"service": "customer", "component": "mysql", "customer_id": "567"}
2020-07-10T03:02:13.299Z INFO frontend/best_eta.go:77      Found customer {"service": "frontend", "customer": {"ID": "567", "Name": "Amazing Coffee Roasters", "Location": "211,653"}}
2020-07-10T03:02:13.300Z INFO driver/client.go:58      Finding nearest drivers
  {"service": "frontend", "component": "driver_client", "location": "211,653"}
2020-07-10T03:02:13.301Z INFO driver/server.go:73      Searching for nearby drivers
  {"service": "driver", "location": "211,653"}
2020-07-10T03:02:13.324Z INFO driver/redis.go:67      Found drivers {"service": "driver", "drivers": ["T732907C", "T791395C", "T705718C", "T724516C", "T782991C", "T703350C", "T771654C", "T724823C", "T718650C", "T785041C"]}
:
```

Logging is very flexible in that it can also be used to infer a lot of information about the application. For example, you can see in the first line of the above log snippet `HTTP request received`, indicating a frontend service request. You can also see log messages related to loading customer information, locating the nearest drivers, etc. There is also a timestamp on each log message so you could calculate the amount of time taken for a particular request by subtracting the ending time from the starting time. You could also calculate the number of requests that were processed in a given interval. To do this type of log analysis at scale, you need cluster-level logging⁵⁵ and a central logging backend like Elasticsearch+Kibana⁵⁶ or Splunk⁵⁷.

Click on a few more of the buttons in the Hot ROD application. We can determine the number of requests by counting the number of `HTTP request received` messages for the frontend service.

```
$ kubectl logs hotrod-59c88cd9c7-sdktk | grep -e "received" | grep frontend | wc -l
7
```

From the above output, we can see that there have been seven frontend requests received since the Pod was started.

However, as critical and flexible as logging is, it is sometimes not the best tool to observe certain aspects of the system. Logs are a very low-level aspect of Observability. Using log

⁵⁵ <https://kubernetes.io/docs/concepts/cluster-administration/logging/>

⁵⁶ <https://www.elastic.co/what-is/elk-stack>

⁵⁷ <https://www.splunk.com/>

messages to derive metrics such as count of requests process, requests per second, etc. can be quite expensive and may not give all the information you need. Also, it is quite tricky, if not impossible, to determine the state of the system at any given moment by parsing log messages without a deep understanding of the code. Often log messages are coming from different threads and sub-processes of the system and must be correlated to one another to follow the system's current state. So, while important, Pod logs are just barely scratching the surface of the observability capabilities of Kubernetes.

In the next section, we will see how Metrics can be used to observe the system's properties instead of low-level log parsing.

Exercise 8.1

Use the `kubectl logs` command to display the Hot ROD Pod's log messages and look for any error messages (Hint: grep for the string `ERROR`). If so, what are the types of errors you see?

8.1.2 Metrics

Another critical aspect of Observability are metrics that measure the performance and operation of the system or application. At a fundamental level, metrics are a set of key-value pairs that provide information about the system's operation. You can think of metrics as the observable properties of each component of the system. Some core resource metrics that apply to all components are CPU, memory, disk, and network utilization. Other metrics may be application-specific, like the number of a particular type of error that has been encountered or the count of items in a queue waiting to be processed.

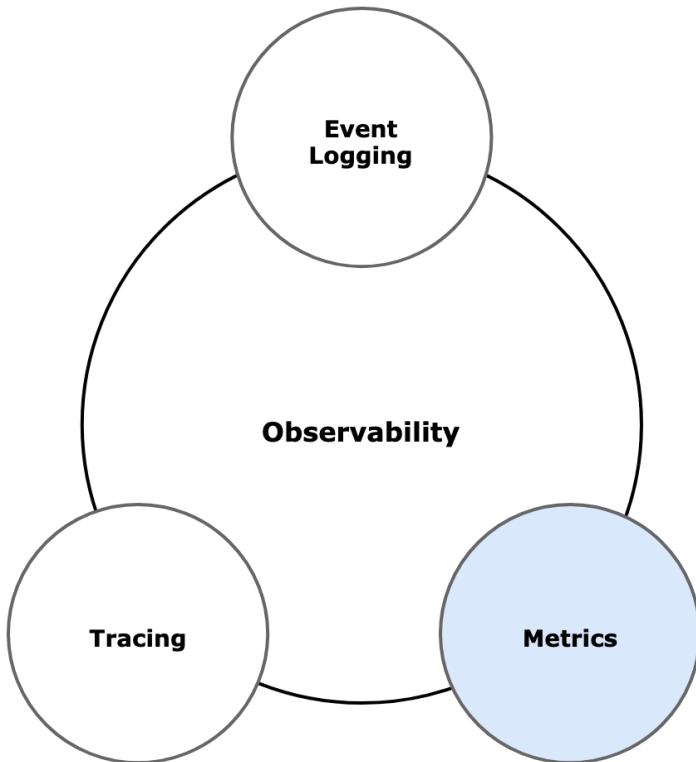


Figure 8.4 Metrics are a set of key-value pairs that provide information about the system's operation.

Kubernetes provides basic metrics using an optional component called the **metrics-server**. The metrics-server can be enabled in minikube by running the following command.

```
$ minikube addons enable metrics-server
* The 'metrics-server' addon is enabled
```

Once the metrics-server is enabled and you wait a few minutes for enough metrics to be collected, you can access the metrics-server data using the commands `kubectl top nodes` and `kubectl top pods`.

Listing 8.2 Output of ‘`kubectl top nodes`’ and ‘`kubectl top pods`’.

```
$ kubectl top nodes
NAME      CPU(cores)   CPU%     MEMORY(bytes)   MEMORY%
minikube  211m        5%       805Mi          40%

$ kubectl top pods --all-namespaces
NAMESPACE      NAME                           CPU(cores)   MEMORY(bytes)
default        hotrod-59c88cd9c7-sdktk        0m          4Mi
kube-system   coredns-66bff467f8-gk4zp       4m          8Mi
```

kube-system	coredns-66bfff467f8-qqxdv	4m	20Mi
kube-system	etcd-minikube	28m	44Mi
kube-system	kube-apiserver-minikube	62m	260Mi
kube-system	kube-controller-manager-minikube	26m	63Mi
kube-system	kube-proxy-vgzw2	0m	22Mi
kube-system	kube-scheduler-minikube	5m	12Mi
kube-system	metrics-server-7bc6d75975-1c5h2	0m	9Mi
kube-system	storage-provisioner	0m	35Mi

The above output shows the CPU and memory utilization of the node (minikube) and the running Pods.

In addition to general CPU and memory utilization that is common across all Pods, applications can provide their own metrics by exposing an HTTP “metrics endpoint” that returns a list of metrics in the form of key-value pairs. Let’s look at the Hot ROD application metrics endpoint that we used in the previous section.

In another terminal, use the `kubectl port-forward` command to forward a port on your workstation to the metrics endpoint of Hot ROD, which is exposed on port 8083 of the Pod.

```
$ kubectl port-forward hotrod-59c88cd9c7-dxd55 8083
Forwarding from 127.0.0.1:8083 -> 8083
Forwarding from [::1]:8083 -> 8083
```

Once the port forward connection is established, open <http://localhost:8083/metrics> in your web browser or run `curl http://localhost:8083/metrics` from the command line.

Listing 8.3 Output of Hot ROD Metrics Endpoint.

```
$ curl http://localhost:8083/metrics
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 6.6081e-05
go_gc_duration_seconds{quantile="0.25"} 8.1335e-05
go_gc_duration_seconds{quantile="0.5"} 0.000141919
go_gc_duration_seconds{quantile="0.75"} 0.000197202
go_gc_duration_seconds{quantile="1"} 0.000371112
go_gc_duration_seconds_sum 0.000993336
go_gc_duration_seconds_count 6
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 26
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.14.4"} 1
:
```

By themselves, metrics provide a snapshot of a system component’s performance and operation at a particular point in time. Often, metrics are collected periodically and stored in a time-series database to observe the metrics’ historical trends. In Kubernetes, this is typically done by a [Cloud Native Computing Foundation \(CNCF\)](#) open-source project called [Prometheus](#).

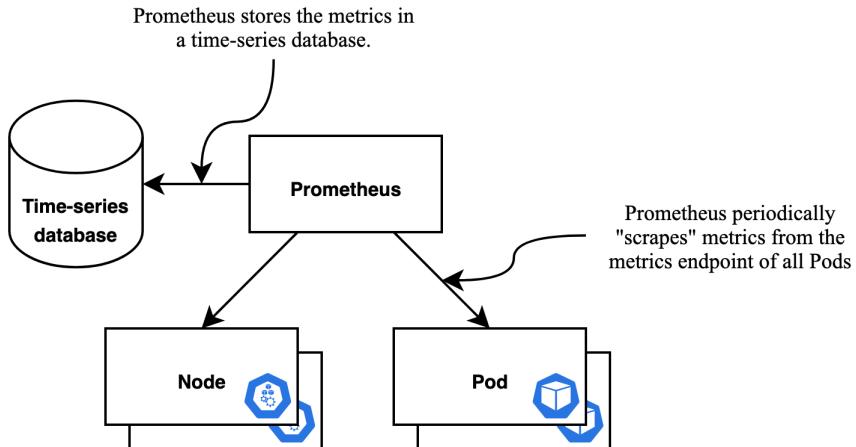


Figure 8.5 A single Prometheus deployment can scrape the metric endpoints of both Nodes and Pods of a cluster. Metrics are scraped at a configurable interval and stored in a time-series database.

As was mentioned earlier in the chapter, some metrics can be inferred through the examination of log messages. Still, it is more efficient to have the system or application measure their metrics directly and provide programmatic access to query the metric values.

Exercise 8.2

Find out the number of different HTTP requests (Hint: search for metrics named "http_requests"). How many `GET /dispatch`, `GET /customer`, and `GET /route` requests have been processed by the application? How would you get similar information from the application's logs?

8.1.3 Tracing

Typically distributed tracing data requires an application-specific agent that knows how to collect the detailed execution paths of the code being traced. A distributed tracing framework captures detailed data of how the system runs internally, from the initial end-user request on through possibly dozens (hundreds?) of calls to different microservices and other external dependencies, perhaps hosted on another system. Whereas metrics typically give an aggregated view of the application in a particular system, tracing data usually provides a detailed picture of an individual request's execution flow, potentially across multiple services and systems. This is particularly important in the age of microservices where an "application" may utilize functionality from tens or hundreds of services and cross multiple operational boundaries. As mentioned earlier in the Logging section, the Hot ROD application is composed of four different microservices (frontend, customer, driver, and route) and two simulated storage backends (MySQL and Redis).

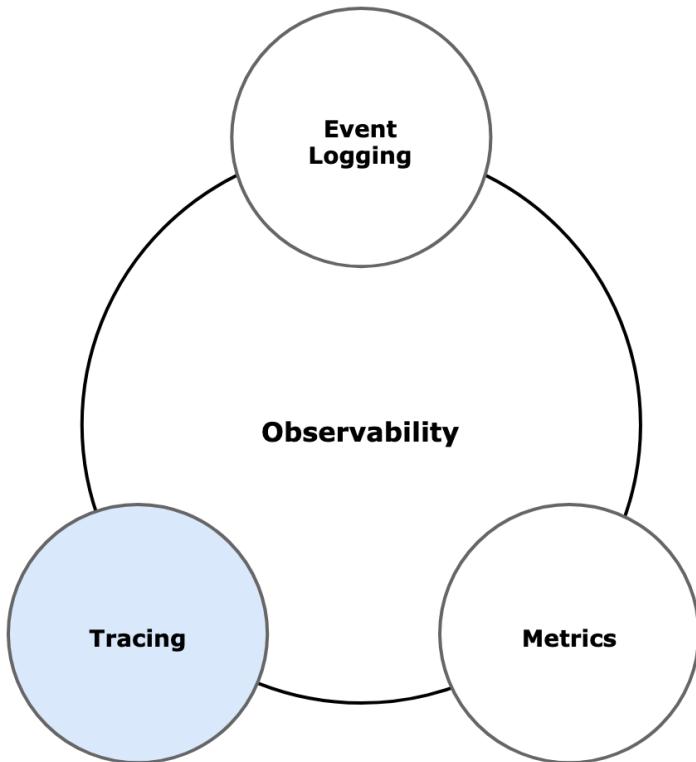


Figure 8.6 Tracing captures detailed data of how the system runs internally.

To illustrate this, let's look at one popular tracing framework, Jaeger, and the example Hot Rod application from the previous Logging and Metrics sections. First, install Jaeger on the minikube cluster and use the following commands to verify it is running successfully.

```

$ kubectl apply -f https://raw.githubusercontent.com/gitopsbook/resources/master/chapter-08/jaeger/jaeger-all-in-one.yaml
deployment.apps/jaeger created
service/jaeger-query created
service/jaeger-collector created
service/jaeger-agent created
service/zipkin created

$ kubectl get pods -l app=jaeger
NAME             READY   STATUS    RESTARTS   AGE
jaeger-f696549b6-f7c9h   1/1     Running   0          2m33s
  
```

Now that Jaeger is running, we need to update the Hot Rod application deployment to send tracing data to Jaeger. This is done simply by adding the `JAEGER_AGENT_HOST` environment variable to the `hotrod` container, indicating the `jaeger-agent` service deployed by `jaeger-all-in-one.yaml` in the above step.

```
$ diff --context=4 hotrod-app.yaml hotrod-app-jaeger.yaml
*** hotrod-app.yaml      2020-07-20 17:57:07.000000000 -0700
--- hotrod-app-jaeger.yaml 2020-07-20 17:57:07.000000000 -0700
*****
*** 22,29 ****
--- 22,32 ---
                    #- "--fix-disable-db-conn-mutex"
                    - "all"
        image: jaegertracing/example-hotrod:latest
        name: hotrod
+
        env:
+
            - name: JAEGER_AGENT_HOST
              value: "jaeger-agent"
+
        ports:
            - name: frontend
              containerPort: 8080
            - name: customer
*****
*** 41,45 ****
--- 44,48 ----
        - port: 80
          targetPort: 8080
        selector:
            app: hotrod
!
        type: LoadBalancer
\ No newline at end of file
--- 44,48 ---
        - port: 80
          targetPort: 8080
        selector:
            app: hotrod
!
        type: LoadBalancer
```

Since we have configured the hotrod-app to send data to Jaeger, we need to generate some trace data by opening the hotrod-app UI and clicking a few buttons as we did in the Logging section.

Use the `minikube service hotrod-frontend` command to create a tunnel on your workstation to the `hotrod-frontend` Service and open the URL in a web browser.

```
$ minikube service hotrod-frontend
|-----|-----|-----|-----|
| NAMESPACE |     NAME      | TARGET PORT |          URL       |
|-----|-----|-----|-----|
| default   | hotrod-frontend |             | http://127.0.0.1:52560 |
|-----|-----|-----|-----|
💡 Opening service default/hotrod-frontend in default browser...
```

This will open a web browser to the application. When it opens, click on each of the buttons to simulate requesting a ride for each customer.

Now that we should have some trace data open the Jaeger UI by running `minikube service jaeger-query`.

```
$ minikube service jaeger-query
|-----|-----|-----|-----|
| NAMESPACE |     NAME      | TARGET PORT |          URL       |
|-----|-----|-----|-----|
| default   | jaeger-query | query-http | http://192.168.99.120:30274 |
|-----|-----|-----|-----|
```

|-----|-----|-----|-----|
 Opening service default/jaeger-query in default browser...

This will open the Jaeger UI in your default browser. Or you can open the URL yourself that is listed in the above output (e.g., <http://127.0.0.1:51831>). When you are finished with the Jaeger exercises in this chapter, you can type Ctrl-C to close the tunnel to the jaeger-query service.

From the Jaeger UI, you can choose the service “frontend” and the operation “HTTP GET /dispatch” and then click “Find Traces” to get a list of all the GET /dispatch call graph traces.

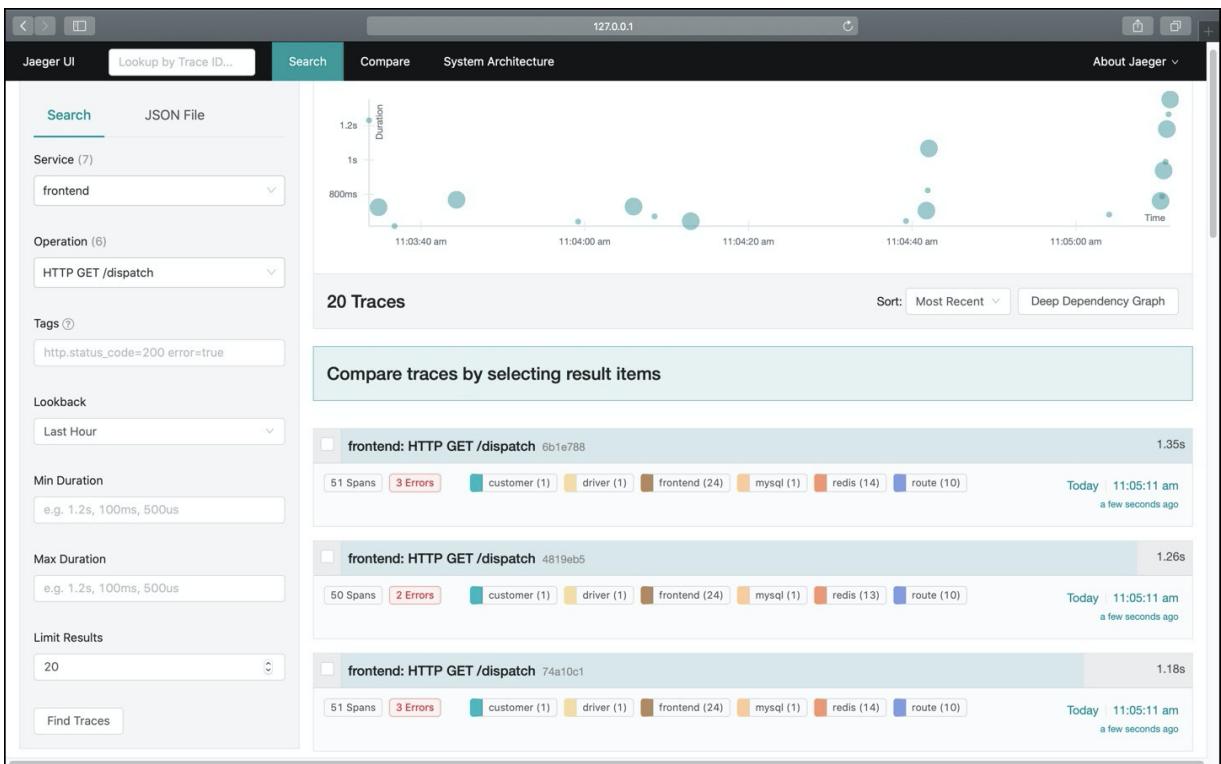


Figure 8.7 The Jaeger search tab displays the GET /dispatch requests from the frontend service that occurred in the last hour. A graph on the top-right shows each request's duration over time, with each circle's size representing the number of spans in each request. The bottom-right lists all the requests and each row can be clicked for additional detail.

From there, you can select the trace to examine. The following screenshot shows a call graph of the frontend GET /dispatch request in the Jaeger UI:

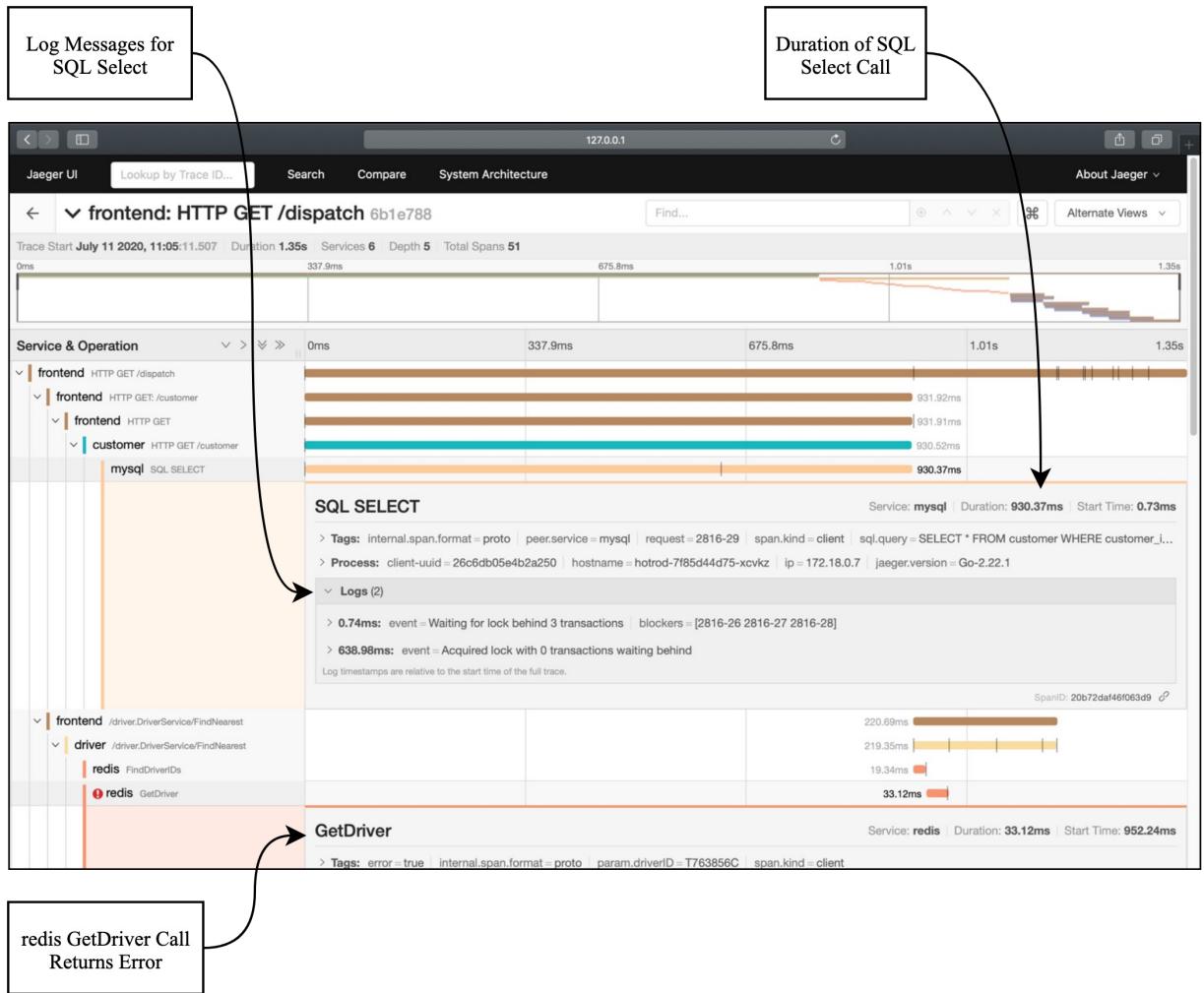


Figure 8.8 In this detailed view of a `GET /dispatch` trace, Jaeger displays all the call spans initiated from the originating request. This example shows such details as the duration and logs of an `SQL SELECT` call and the Redis `GetDriver` call's return error.

As you can see from the above screenshot, there is lots of valuable information regarding the `GET /dispatch` request processing. From this, you can see the breakdown of what code is being called, what its response is (success or failure), and how long it took. For example, in the above example screenshot, it appears that the SQL query used by this request took 930.37ms. Is that good? The application developer can do more testing and dig deeper to see if that query can be optimized or if there is a different area of the code that would benefit from additional optimization.

Again, as was mentioned earlier, a developer may sprinkle their code with log statements to have “tracing data” in their application logs, but this is a costly and inefficient approach. Using a proper framework for tracing is much more desirable and will provide a much better result in the long run.

As you can imagine, tracing data can be quite large, and it may not be feasible to collect the data for every single request. Tracing data is often sampled at a configured rate and may have a much shorter retention period than, for example, application logs or metrics.

But what does tracing have to do with GitOps? Honestly, not much. But tracing is an essential and growing part of Observability, with many new developments in tools and services that help provide, manage, and analyze distributed tracing data, so it’s important to understand how it fits in the overall Observability system. It is also possible that in the future, tracing tools (e.g., OpenTelemetry) will be used for more aspects of Observability by expanding to cover metrics and logging.

Exercise 8.3

Identify the performance bottleneck in Hot ROD using the Jaeger distributed tracing platform.⁵⁸ To do this, open the Hot ROD UI in a browser window. Click a few customers in the Hot ROD UI (e.g., Japanese Desserts and Amazing Coffee Roasters) to schedule rides in the application. Actually, go crazy! Keep clicking the different customer buttons a bunch of times.

Once you’ve played around a bit with the application, open the Jaeger UI as described earlier in this section. Use the search capabilities to find traces for various services and requests to answer the questions below. You may want to change the “Limit Results” to a larger value, like maybe 200.

1. Do you have any traces containing errors? If so, what component is causing the errors?
2. Do you notice any difference in latency in requests when you click customer buttons slowly vs. very quickly? Do you have any dispatch requests that take longer than 1000ms?
3. Search Jaeger for the trace with the longest latency according to the Hot ROD app using the driver ID (which is in bold). Hint: Use the “Tags” filter and search for “driver=T123456C”, where “T123456C” is the driver ID of your longest latency request.
4. Where is the application spending the most time? What span is the longest?
5. What do the logs of the longest span say? Hint: The log message starts with Waiting for....
6. Based on what you discovered in the previous question, what might be impacting the performance of the Hot ROD application? Examine the following code:

<https://github.com/jaegertracing/jaeger/blob/master/examples/hotrod/services/customer/database.go#L86-L90>

⁵⁸ <https://github.com/jaegertracing/jaeger/tree/master/examples/hotrod>

Exercise 8.3

Fix the Hot ROD application's performance bottleneck by adding the `--fix-disable-db-conn-mutex` argument to the `hotrod` container and updating the deployment. (Hint: Look at the Github [hotrod-app-jaeger.yaml](#) file and uncomment the appropriate line.) This simulates fixing a database lock contention in the code.

1. Update the `hotrod` container by adding the `--fix-disable-db-conn-mutex` argument
2. Redeploy the updated `hotrod-app-jaeger.yaml` file.
3. Test the `hotrod` UI. Do you notice a difference in the latency of each dispatch? Can you get a dispatch request to take longer than 1000ms?
4. Look at the Jaeger UI. Do you notice the difference in the traces? What is different after adding the `--fix-disable-db-conn-mutex` argument?

To dive deeper into Jaeger and the Hot ROD sample application, refer to the blog and video links at the top of the [README page](#).⁵⁹

8.1.4 Visualization

All of the above aspects of observability are really about the system providing data about itself. And it adds up to a lot of data that can be difficult to make sense of. The last aspect of observability is visualization tools that help convert all that observability data into information and insights.

Many tools provide visualization of observability data. In the previous section, we discussed Jaeger, which provides visualization of trace data. But now, let's look at another tool that provides visualization of the current running state of your Kubernetes cluster, the K8s Dashboard.

You can enable the K8s Dashboard minikube addon⁶⁰ using the following command.

```
$ minikube stop #A
✋ Stopping "minikube" in docker ...
● Powering off "minikube" via SSH ...
● Node "minikube" stopped.

$ minikube addons enable dashboard #B
★ The 'dashboard' addon is enabled
```

#A Stop minikube if it is already running

#B Enable the Dashboard addon

Once enabled, you can start your minikube cluster and display the Dashboard.

```
$ minikube start
🕒 minikube v1.11.0 on Darwin 10.15.5
leftrightarrow Using the docker driver based on existing profile
👉 Starting control plane node minikube in cluster minikube
💡 minikube 1.12.0 is available! Download it:
      https://github.com/kubernetes/minikube/releases/tag/v1.12.0
❗ To disable this notice, run: 'minikube config set WantUpdateNotification false'
```

⁵⁹ <https://github.com/jaegertracing/jaeger/blob/master/examples/hotrod/README.md>

⁶⁰ <https://minikube.sigs.k8s.io/docs/tasks/addons/>

```
⌚ Restarting existing docker container for "minikube" ...
⌚ Preparing Kubernetes v1.18.3 on Docker 19.03.2 ...
  ▪ kubeadm.pod-network-cidr=10.244.0.0/16
🔑 Verifying Kubernetes components...
🌟 Enabled addons: dashboard, default-storageclass, metrics-server, storage-provisioner
  #A
🏃 Done! kubectl is now configured to use "minikube"
$ minikube dashboard
```

#A Dashboard addon is started with minikube

The command `minikube dashboard` will open a browser window that looks similar to this:



Figure 8.9 The Overview page of the Kubernetes Dashboard shows the CPU and Memory usages of the workloads currently running on the cluster, along with a summary chart of each type of workload (Deployments, Pods, and Replica Sets). The page's bottom-right pane shows a list of each workload `GET /dispatch`, and each row can be clicked for additional detail.

The Kubernetes dashboard provides visualization of many different aspects of your Kubernetes cluster, including the status of your Deployments, Pods, and Replica Sets. We can see that there is a problem with the `bigpod` Deployment. Upon seeing this, the Kubernetes administrator should take action to bring this Deployment back into a healthy state.

Exercise 8.4

Install the Dashboard addon on minikube. Open the Dashboard UI, explore the different panels available, and perform the following actions:

1. Switch the view to "All Namespaces." How many total Pods are running on Minikube?
2. Select the Pod panel. Filter the Pods list to those containing "dns" in the name by clicking on the filter icon in the top-right of the Pod panel. Delete one of the DNS-related pods by clicking the action icon for that Pod and choosing "Delete." What happens?
3. Click on the plus icon ("+") in the top-right of the Dashboard UI and select "Create from form." Create a new nginx deployment containing 3 Pods using the nginx image. Experiment with the "Create from input" and "Create from file" options, perhaps using code listings from earlier chapters.

8.1.5 Importance of Observability in GitOps

Okay, so now you know what Observability is and that it is generally a good thing, but you may wonder why a book on GitOps would devote a whole chapter on Observability. What does Observability have to do with GitOps?

As was discussed in chapter 2, using the declarative configuration of Kubernetes allows the desired state of the system to be precisely defined. But how do you know if the running state of the system has converged with the desired state? How do you know if a particular deployment was successful? More broadly, how can you tell if your system is working as intended? These are critical questions that GitOps and observability should help answer.

There are several aspects of observability in Kubernetes that are critical for the GitOps system to function well and answer critical questions about the system:

Application Health: Is the application operating correctly? If a new version of the application is deploying using GitOps, is the system "better" than before?

Application Sync Status: Is the running state of the application the same as the desired state defined in the deployment Git repo?

Configuration Drift: Has the application's configuration been changed outside of the declarative GitOps system (e.g., either manually or imperatively)?

GitOps Change Log: What changes were recently made to the GitOps system? Who made them, and for what reason?

The remainder of this chapter will cover how the Observability of the Kubernetes system and the application allow the above questions to be answered by the GitOps system and how, in turn, the GitOps system provides observability capabilities.

8.2 Application Health

The first and most important way observability relates to GitOps is the ability to observe application health. At the beginning of the chapter, we talked about how operating a system is really about managing that system so that overall it improves and gets better over time instead of getting worse. GitOps is a key part of that where the desired state of the system (one that presumably is "better" than the current state) is committed to the Git repo, and

then the GitOps Operator applied that desired state to the system, making it become the current state.

For example, imagine that when the Hot ROD application discussed earlier in this chapter was initially deployed, it had relatively small operating requirements. However, over time, as the application becomes more popular, the data set grows and the memory allocated to the Pod is no longer enough. The Pod periodically runs out of memory and is terminated (an event called “OOMKilled”). The application health of the Hot ROD application would show that it is periodically crashing and restarting. The system operator could check-in a change to the Git deployment repository for the Hot ROD application that increases its requested memory. The GitOps Operator would then apply that change, increasing the memory of the running Hot ROD application.

Perhaps we could leave things there. After all, someone committed a change to the system, and GitOps should just do what it’s been told. But what if the committed change actually makes things worse? What if the application comes back up after deploying the latest change but then starts returning more errors than it did before? Or even worse, what if it doesn’t come back up at all? What if, in the above example, the operator increased the memory for the Hot ROD application too much, causing other applications running on the cluster to run out of memory?

With GitOps, we would at least like to detect those conditions where, after a deployment, the system is “worse” than before and, at a minimum, alert the users that perhaps they should consider rolling back the most recent change. This is only possible if the system and application have strong observability characteristics that the GitOps operator can, well, observe.

8.2.1 Resource Status

At a basic level, a key feature of Kubernetes that enables observability of its internal state is how, with a declarative configuration, the desired configuration and active state are both maintained. This allows every Kubernetes resource to be inspected at any time to see whether or not the resource is in the desired state. Each component or resource will be in a particular operational state at any given time. For example, a resource might be in INITIALIZED or NOT READY state. It also may be in a PENDING, RUNNING, or COMPLETED state. Often the status of a resource is specific to its type.

The first aspect of application health is determining that all the Kubernetes resources related to the application are in a good state. For example, have all the Pods been successfully scheduled and are in Running state? Let’s take a look at how this can be determined.

Kubernetes provides additional information for each pod that indicates if a pod is healthy or not. Let’s run the `kubectl describe` command on the etcd pod.

```
$ kubectl describe pod etcd-minikube -n kube-system
Name:           etcd-minikube
Namespace:      kube-system
Priority:      2000000000
Priority Class Name: system-cluster-critical
Node:          minikube/192.168.99.103
Start Time:    Mon, 10 Feb 2020 22:54:14 -0800
```

```

:
Status:           Running
IP:              192.168.99.103
IPs:
  IP:          192.168.99.103
Controlled By: Node/minikube
Containers:
  etcd:
    :
    Command:
      :
    State:        Running
    Started:     Mon, 10 Feb 2020 22:54:05 -0800
    Ready:        True
    Restart Count: 0
    Liveness:     http-get http://127.0.0.1:2381/health delay=15s timeout=15s period=10s
                  #success=1 #failure=8
    Environment:  <none>
    Mounts:
      :
Conditions:
  Type            Status
  Initialized     True
  Ready           True
  ContainersReady True
  PodScheduled    True
Volumes:
  :
QoS Class:      BestEffort
Node-Selectors:  <none>
Tolerations:    :NoExecute
Events:         <none>

```

The above output has been truncated for brevity, but take a closer look at it and compare it to the output from your minikube. Do you see any properties of the Pod that may help with observability? One very important (and obvious) one is towards the top is `Status: Running`, which indicates the phase the Pod is in. The Pod phase is a simple, high-level summary of where the Pod is in its life cycle. The conditions array, the reason and message fields, and the individual container status arrays contain more detail about the pod's status.

Table 8.1 Pod Phases

Phase Values	Description
Pending	The Kubernetes system has accepted the pod, but one or more container images have not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while
Running	The pod has been bound to a node, and all of the containers have been created. At least one container is still running or is in the process of starting or restarting.

Succeeded	All containers in the pod have terminated in success and will not be restarted.
Failed	All containers in the pod have completed, and at least one container has terminated in failure. The container either exited with non-zero status or was terminated by the system.
Unknown	For some reason, the pod state could not be obtained, typically due to an error in communicating with the pod's host.

This is observability; the internal state of a Pod can be easily queried so decisions can be made about controlling the system. For GitOps, if a new version of an application is deployed, it is important to look at the resulting new Pods state to ensure its success.

But the Pod state (phase) is only a summary, and, to understand why a Pod is in a particular state, you need to look at the Conditions. A Pod has four conditions that will be either True, False, or Unknown.

Table 8.2 Pod Conditions

Phase Values	Description
PodScheduled	The Pod has been successfully scheduled to a node in the cluster.
Initialized	All init containers have started successfully.
ContainersReady	All containers in the Pod are ready.
Ready	The Pod can serve requests and should be added to the load balancing pools of all matching Services.

Exercise 8.5

Use the `kubectl describe` command to display the information of other pods running in the `kube-system` namespace.

An elementary example of how a Pod may get into a bad state is submitting a Pod with a manifest that requests more resources than are available in the cluster. This will cause the Pod to go into `Pending` state. The status of the Pod can be “observed” by running `kubectl describe pod <pod_name>`.

Listing 8.4 `bigpod.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: bigpod
spec:
  containers:
    - command:
        - /app/sample-app
      image: gitopsbook/sample-app:v0.1
      name: sample-app
      ports:
        - containerPort: 8080
      resources:
        requests:
          memory: "999Gi"                                #A
          cpu: "99"                                     #B

```

#A Request an impossible amount of memory

#B Request an impossible number of CPU cores

If you apply this yaml and check Pod status, you will notice that the Pod is Pending. When you run the `kubectl describe`, you will see that the Pod is in Pending state since the minikube cluster can't satisfy the resource request for 999 GB of RAM, nor the request for 99 CPUs.

```

$ kubectl apply -f bigpod.yaml
deployment.apps/bigpod created

$ kubectl get pod
NAME                  READY   STATUS    RESTARTS   AGE
bigpod-7848f56795-hnpjx   0/1     Pending   0          5m41s

$ kubectl describe pod bigpod-7848f56795-hnpjx
Name:           bigpod-7848f56795-hnpjx
Namespace:      default
Priority:       0
Node:           <none>
Labels:          app=bigpod
                 pod-template-hash=7848f56795
Annotations:    <none>
Status:         Pending                                         #A
IP:
IPs:            <none>
Controlled By: ReplicaSet/bigpod-7848f56795
Containers:
  bigpod:
    Image:      gitopsbook/sample-app:v0.1
    Port:       8080/TCP
    Host Port:  0/TCP
    Command:
      /app/sample-app
    Requests:
      cpu:        99
      memory:    999Gi
    Environment: <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-8dzwz (ro)
Conditions:
  Type      Status
  PodScheduled  False                                         #B
Volumes:
```

```

default-token-8dzwz:
  Type:          Secret (a volume populated by a Secret)
  SecretName:   default-token-8dzwz
  Optional:     false
  QoS Class:    Burstable
  Node-Selectors: <none>
  Tolerations:   node.kubernetes.io/not-ready:NoExecute for 300s
                  node.kubernetes.io/unreachable:NoExecute for 300s
  Events:
    Type      Reason           Age             From            Message
    ----      -----          ----           ----           -----
    Warning   FailedScheduling <unknown>       default-scheduler 0/1 nodes are available: 1
              Insufficient cpu, 1 Insufficient memory.
    Warning   FailedScheduling <unknown>       default-scheduler 0/1 nodes are available: 1
              Insufficient cpu, 1 Insufficient memory.

```

#A Pod status is Pending

#B PodScheduled condition is False

#C No node is available with sufficient memory or CPU to schedule the Pod

Exercise 8.6

Update the `bigpod.yaml` with a more reasonable request for CPU and memory and redeploy the Pod. (Hint: Change CPU to `99m` and memory to `999Ki`.) Run `kubectl describe` on the updated Pod and compare the output with the output before your changes. What is the Status, Conditions, and Events of the updated Pod?

8.2.2 Readiness and Liveness

If you look carefully at the Pod Conditions listed in Table 8.2, something might stand out. The `Ready` state claims that “the Pod can serve requests.” How does it know? What if the Pod has to perform some initialization? How does Kubernetes know that the Pod is ready? The answer is that the Pod itself notifies Kubernetes when it is ready based on its own application-specific logic.

Kubernetes uses Readiness probes to decide when the Pod is available for accepting traffic. Each container in the Pod can specify a Readiness probe, in the form of a command or an HTTP request, that will indicate when the container is `Ready`. It is up to the container to provide this observability about its internal state. Once all the Pod containers are `Ready`, then the Pod itself is considered `Ready` and can be added to load balancers of matching Services and begin handling requests.

Similarly, each container can specify a Liveness probe that indicates if the container is alive and not, for example, in some sort of deadlock situation. Kubernetes uses Liveness probes to know when to restart a container that has entered a bad state.

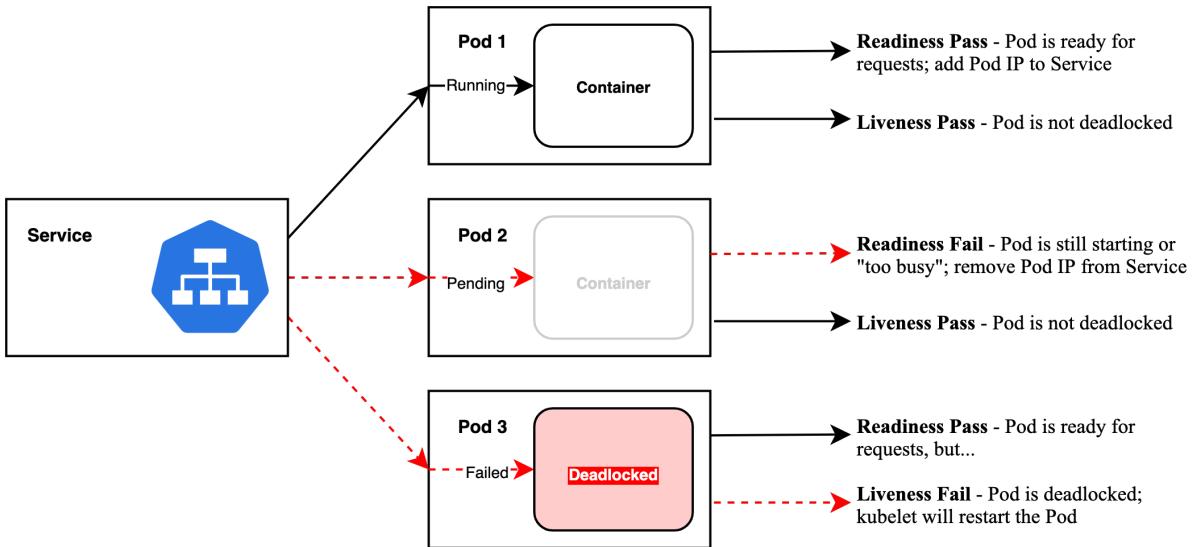


Figure 8.10 Kubernetes uses Readiness and Liveness Probes to determine which Pods are available to accept traffic. Pod 1 is in Running state, and both the Readiness and Liveness probes are passing. Pod 2 is in Pending state and, while the Liveness probe is passing, the Readiness probe is not since the Pod is still starting up. Pod 3 passes the Readiness probe but fails the Liveness probe, meaning it will likely soon be restarted by kubelet.

So here again is an aspect of Observability that is built into Kubernetes. Application pods provide visibility to their internal state through Readiness and Liveness probes so that the Kubernetes system can decide how to control them. Application developers must properly implement these probes so that the application provides the correct observability of its operation.

Exercise 8.7

Create a pod spec that uses an init container to create a file and configure the Liveness and Readiness probes of the app container to expect that the file exists. Create the Pod and then see its behavior. Exec into the Pod to delete the file and see its behavior.

8.2.3 Application Monitoring and Alerting

In addition to status and readiness/liveness, applications typically have vital metrics that can be used to determine their overall health. This is the foundation of operational monitoring and alerting: watch a set of metrics and set off alarms when they deviate from allowable values. But what metrics should be monitored, and what are the permissible values?

Fortunately, there has been a lot of research done on this topic, and it has been well covered in other books and articles. Rob Ewaschuk described the "four golden signals" as the

most important metrics to focus on at a high-level. This provides a useful framework for thinking about metrics.⁶¹

- Latency - The time it takes to service a request.
- Traffic - A measure of how much demand is placed on the system.
- Errors - The rate of requests that are not successful.
- Saturation - How "full" your service is.

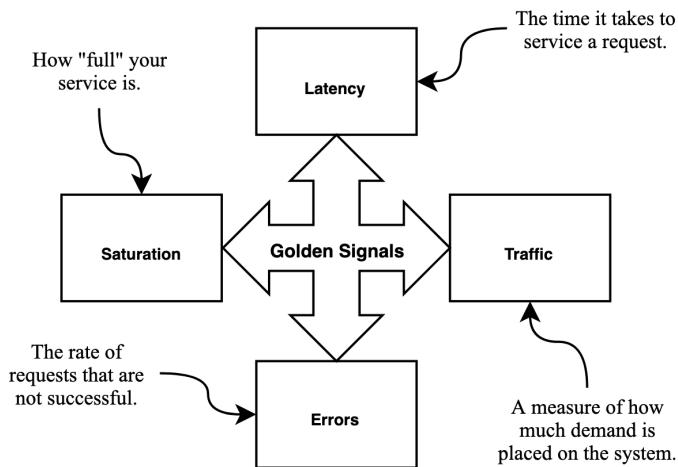


Figure 8.11 The four “Golden Signals,” latency, traffic, errors, and saturation, are the critical metrics to measure that indicate the system’s overall health. Each measures a specific operational aspect of the system. Any given problem in the system is likely to manifest itself by adversely impacting one or more of the “Golden Signal” metrics.

Brendan Gregg proposed the USE Method for characterizing the performance of system resources (like infrastructure, such as Kubernetes nodes)⁶².

- Utilization - The average time that the resource was busy servicing work.
- Saturation - The degree to which the resource has extra work which it can't service, often queued.
- Errors - The count of error events.

For request-driven applications (like microservices), Tom Wilkie defined the RED Method⁶³.

- Rate - The number of requests per second a service is processing
- Errors - The number of failed requests per second
- Duration - Distributions of the amount of time each request takes.

⁶¹ <https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/>

⁶² <http://www.brendangregg.com/usemethod.html>

⁶³ <https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/>

While a more in-depth discussion of determining application health through metrics is beyond this book's scope, we highly recommend reading the three related footnoted references summarized above.

Once you have identified the metrics to evaluate application health, you need to monitor them and generate alerts when they fall outside of allowable values. In traditional operational environments, this would typically be done by a human operator staring at a dashboard, but perhaps by an automated system. If the monitoring detects an issue, an alert is raised, triggering the on-call engineer to look at the system. The on-call engineer analyses the system and determines the correct course of action to resolve the alert. This might be to stop the rollout of a new release to the fleet of servers or perhaps even roll back to the previous version.

All this takes time and delays the recovery of the system back to an optimal running state. What if the GitOps system could help improve this situation?

Consider the case where all the pods come up successfully. All the readiness checks are successful, but once the application begins processing, the length of time required to process each request suddenly increases by 2x (the Duration of the RED Method). It may be that a recent code change introduced a performance bug that is degrading the performance of the application.

Ideally, such a performance issue should be caught while testing in pre-production. If not, might it be possible for the GitOps operator and deployment mechanism to automatically stop or roll back the deployment if particular "golden signal" metrics suddenly become degraded and deviate from an established baseline?

This is covered in more detail as part of a discussion of advanced observability use-cases in Section 8.6.

8.3 GitOps Observability

Often administrators will change the system configuration defined in GitOps based on observed application health characteristics. If a Pod is stuck in `CrashLoopBackoff` state due to an out of memory condition, the Pod's manifest may be updated to request more memory for the Pod. If a memory leak causes the out of memory condition in the application, perhaps the Pod's image will be updated to one that contains a fix to the memory leak. Maybe the "golden signals" of the application indicate that it is reaching saturation and cannot handle the load, so the Pod manifest may be updated to request more CPU, or the number of replicas of the Pod is increased to horizontally scale the application.

These are all GitOps operations that would be taken based on the observability of the application. But what about the GitOps process itself? What are the observable characteristics of a GitOps deployment?

8.3.1 GitOps Metrics

If the GitOps operator or service is an application, what are its "golden signals"? Let's consider each area to understand some of the observability characteristics provided by the GitOps operator.

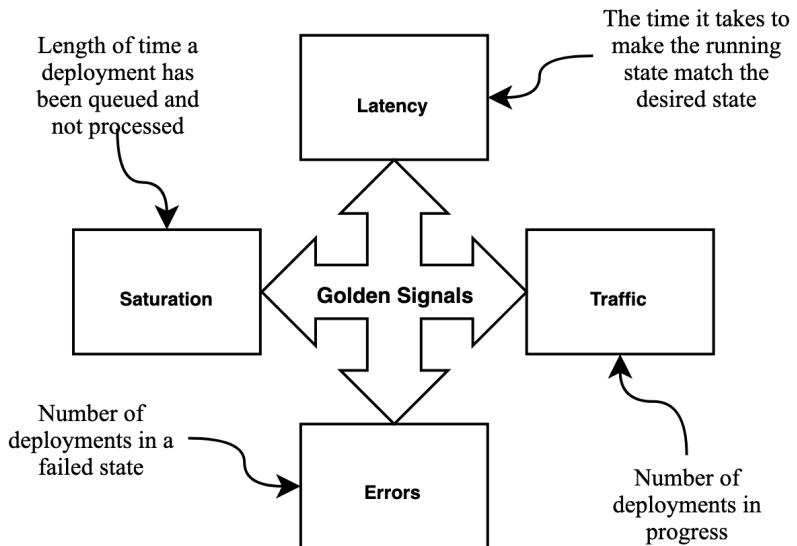


Figure 8.12 The four “Golden Signals” for GitOps indicate the health of the GitOps Continuous Deployment system. Any issue with the GitOps Operator/Service will likely manifest itself by adversely impacting one or more of these “Golden Signal” metrics.

Latency

- The time it takes to deploy and make the system's running state match its desired state.

Traffic

- Frequency of deployments
- The number of deployments in progress.

Errors

- The number of deployments that failed and the current number of deployments in a failed state.
- The number of “out of sync” deployments where the system's running state does not match its desired state.

Saturation

- Length of time a deployment has been queued and not processed.

The implementation of each of these metrics and how they are exposed will be different for each GitOps tool. This will be covered in more detail in the “Tools” chapters of this book.

8.3.2 Application Sync Status

The most important status the GitOps GitOps operator must provide is whether the desired state of the application in the Git repo is the same as the current state of the application (in-sync) or not (out-of-sync). If the application is out-of-sync, the user should be alerted that a deployment (or redeployment) may be needed.

But what would cause an application to become out-of-sync? This is part of the regular operation of GitOps; a user commits a change to the desired state of the system. At the moment that change is committed, the current state of the application doesn't match the desired state.

Let's consider how the "Basic GitOps Operator" described in Section 2.5.1 of Chapter 2 functioned. In that example, a CronJob periodically runs the "Basic GitOps Operator" so that the repository is checked out, and the manifests contained in the repo are automatically applied to the running system.

In this basic example of a GitOps Operator, the question of application sync status is entirely side-stepped for the sake of simplifying the example. The Basic GitOps Operator assumed that the application was out-of-sync at every scheduled execution (or polling interval) and needed to be deployed. This simplistic approach is not suitable for real-world production use because the user has no visibility into whether changes exist that need to be deployed or what those changes are. It might also add unnecessary additional load on the GitOps Operation, Git server, and the Kubernetes API Server.

SAMPLE APPLICATION

Let's run through several different deployment scenarios with our sample-app to explore other Application Sync Status aspects. The sample-app is a simple Golang application that returns an HTTP response message of "Kubernetes ❤ <input>".

First, login to www.github.com and fork the <https://github.com/gitopsbook/sample-app-deployment> repository. If you have previously forked this repo, it's recommended to delete the old fork and re-fork to be sure to start with a clean repository without any changes from previous exercises.

After forking, clone your fork repository using the following command:

```
$ git clone git@github.com:<username>/sample-app-deployment.git
$ cd sample-app-deployment
```

Let's manually deploy the sample-app to minikube:

```
$ minikube start
🕒 minikube v1.11.0 on Darwin 10.14.6
⚡ Automatically selected the hyperkit driver
👉 Starting control plane node minikube in cluster minikube
🔥 Creating hyperkit VM (CPUs=2, Memory=2200MB, Disk=20000MB) ...
⌚ Preparing Kubernetes v1.16.10 on Docker 19.03.8 ...
🔎 Verifying Kubernetes components...
🌟 Enabled addons: default-storageclass, storage-provisioner
🏁 Done! kubectl is now configured to use "minikube"
$ kubectl create ns sample-app
namespace/sample-app created
$ kubectl apply -f . -n sample-app
```

```
deployment.apps/sample-app created
service/sample-app created
```

NOTE: At the time of this writing, Kubernetes v1.18.3 reports additional differences using the `kubectl diff` command. If you experience this issue while completing the following exercises, you can start minikube with an older version of Kubernetes using the command `minikube start --kubernetes-version=1.16.10`.

DETECTING DIFFERENCES

Now that the sample-app has been successfully deployed, let's make some changes to the deployment. Let's increase the number of replicas of the sample-app to 3 in the `deployment.yaml` file. Use the following command to change the replica count of the Deployment resource.

```
$ sed -i '' 's/replicas: .*/replicas: 3/' deployment.yaml
```

Or you could use your favorite text editor to change `replicas: 1` to `replicas: 3` in the `deployment.yaml` file. Once the change has been made to `deployment.yaml`, run the following command to see the uncommitted differences in your Git fork repository.

```
$ git diff
diff --git a/deployment.yaml b/deployment.yaml
index 5fc3833..ed2398a 100644
--- a/deployment.yaml
+++ b/deployment.yaml
@@ -3,7 +3,7 @@ kind: Deployment
 metadata:
   name: sample-app
 spec:
-  replicas: 1
+  replicas: 3
   revisionHistoryLimit: 3
   selector:
     matchLabels:
```

Finally, use `git commit` and `git push` to push changes to the remote Git fork repository:

```
$ git commit -am "update replica count"
[master 5a03ca3] update replica count
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 353 bytes | 353.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:telenstam/sample-app-deployment.git
 09d6663..5a03ca3  master -> master
```

Now that you have committed a change to the sample-app deployment repo, the sample-app GitOps sync status is “out-of-sync” because the current state of the Deployment still only has three replicas. Let’s just confirm that this is the case.

```
$ kubectl get deployment sample-app -n sample-app
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
sample-app  1/1     1           1           3m27s
```

From the above command, you see there are 1/1 READY replicas for the sample-app Deployment. But is there a better way to compare the deployment repo, which represents the desired state, against the running application’s actual state? Luckily Kubernetes provides tools for detecting differences.

KUBECTL DIFF

Kubernetes provides the kubectl diff command, which takes a file or directory as input, and displays the differences between the resources defined in those files and the current resources of the same names in the Kubernetes cluster. If we run kubectl diff against our existing deployment repo, we see the following:

```
$ kubectl diff -f . -n sample-app
@@ -6,7 +6,7 @@
  creationTimestamp: "2020-06-01T04:17:28Z"
- generation: 2
+ generation: 3
  name: sample-app
  namespace: sample-app
  resourceVersion: "2291"
@@ -14,7 +14,7 @@
  uid: eda45dca-ff29-444c-a6fc-5134302bcd81
  spec:
    progressDeadlineSeconds: 600
-  replicas: 1
+  replicas: 3
    revisionHistoryLimit: 3
    selector:
      matchLabels:
```

From the above output, we can see that kubectl diff correctly identified that replicas was changed from 1 to 3.

While this is an elementary example to illustrate the point, this same technique can identify more extensive changes across multiple different resources. This gives the GitOps operator or service the ability to determine when the deployment repo containing the desired state in Git is out-of-sync with the current live state of the Kubernetes cluster. More importantly, the kubectl diff output provides a preview of the changes that would be applied to the cluster if the deployment repo is synced. This is a crucial feature of GitOps observability.

Exercise 8.8

Make a fork of the sample-app deployment repository. Deploy the sample-app to your minikube cluster as described in the sample-app-deployment README.md. Now change the sample-app service to be of type: LoadBalancer. Run the kubectl diff -f . -n sample-

app command. Do you see any unexpected changes? Why? Apply the changes using kubectl apply -f . -n sample-app. Now you should see the sample-app web page using the command minikube service sample-app -n sample-app.

KUBEDIFF

In the above sections, we covered how to see the differences between revisions in the Git repository using git diff and the differences between the Git repository and the live Kubernetes cluster using kubectl diff. In both cases, the diff tools give you a very raw view of the differences, outputting lines before and after the differences for context. And kubectl diff may also report differences that are system managed (like generation) and not relevant to the GitOps use-case. Wouldn't it be cool if there was a tool that gave you a concise report of each resource's specific attributes that are different? As it turns out, the folks at Weaveworks⁶⁴ have released an open-source tool called kubediff⁶⁵ that does precisely that.

Here is the output of kubediff run against our deployment repo of the sample-app:

```
$ kubediff --namespace sample-app .
## sample-app/sample-app (Deployment.v1.apps)

.spec.replicas: '3' != '1'
```

kubediff can also output JSON structured output, allowing it to be more easily used programmatically. Here is the same command run with JSON output:

```
$ kubediff --namespace sample-app . --json
{
  "./deployment.yaml": [
    ".spec.replicas: '3' != '1'"
  ]
}
```

Exercise 8.9

Run kubediff against the sample-app-deployment repository. If not installed already in your environment, you will first need to install Python and pip and run pip install -r requirements.txt as described in the kubediff README.

8.3.3 Configuration Drift

But how else could an application become out-of-sync with the desired state defined in the Git repo? Possibly a user modified the running application directly (like by performing a kubectl edit on the Deployment resource) instead of committing the desired change to the Git repo. We call this "configuration drift."

This is usually a big "no-no" when managing a system with GitOps; you should avoid directly modifying the system outside of GitOps. For example, if your pods are running out of

⁶⁴ <https://www.weave.works/>

⁶⁵ <https://github.com/weaveworks/kubediff>

capacity, one might just simply perform a `kubectl edit` to increase the replica count to increase capacity.

This situation sometimes happens. When it does, the GitOps operator will need to “observe” the current state and detect a difference with the desired state and indicate to the user that the application is “out-of-sync.” A particularly aggressive GitOps operator might automatically redeploy the last previously deployed configuration, thereby overwriting the manual changes.

Using minikube and the sample-app-deployment repository we’ve been using for the last few sections, run `kubectl apply -f . -n sample-app` to make sure the current contents are deployed to Kubernetes. Now run `kubectl diff -f . -n sample-app`; you should see no differences.

Now, let’s simulate a change being made to the application deployment outside of the GitOps system by running the following command:

```
$ kubectl scale deployment --replicas=4 sample-app -n sample-app
deployment.apps/sample-app scaled
```

Now, if we rerun the `kubectl diff` command, we see that the application is out-of-sync, and we have experienced “configuration drift.”

```
$ kubectl diff -f . -n sample-app
@@ -6,7 +6,7 @@
  creationTimestamp: "2020-06-01T04:17:28Z"
-
+ generation: 6
+ generation: 7
  name: sample-app
  namespace: sample-app
  resourceVersion: "16468"
@@ -14,7 +14,7 @@
  uid: eda45dca-ff29-444c-a6fc-5134302bcd81
spec:
  progressDeadlineSeconds: 600
-
+ replicas: 4
+ replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
```

Or if you run `kubediff`, you see the following:

```
$ kubediff --namespace sample-app .
## sample-app/sample-app (Deployment.v1.apps)

.spec.replicas: '3' != '4'
```

Configuration drift is very similar to an application being out-of-sync. In fact, as you see, the effect is the same; the current live state of the configuration is different from the desired configuration as defined in the Git deployment repo. The difference is that typically an application is out-of-sync when a new version is committed to the Git deployment repo that hasn’t been deployed yet. In contrast, configuration drift happens when changes to the configuration have been made outside of the GitOps system.

In general, one of two things must occur when configuration drift is encountered. Some systems would consider the configuration drift an error state and allow a self-healing process to be initiated to sync the system back to the declared state. Other systems may detect this drift and allow the manual change to be integrated back into the declared state saved in Git (i.e., 2-way syncing). However, our view is that 2-way syncing is not desirable because it allows and encourages manual changes to the cluster and bypasses the security and review process that GitOps provides as one of its core benefits.

Exercise 8.10

From the sample-app-deployment, run the command `kubectl delete -f . -n sample-app`. Oops, you just deleted your application! Run `kubectl diff -f . -n sample-app`. What differences do you see? How can you restore your application to a running state? Hint: it should be easy.

8.3.4 GitOps Change Log

Earlier in this chapter, we discussed how Event Logs are a key aspect of Observability. For GitOps, the “Event Log” of the application deployment is primarily composed of the deployment repository’s commit history. Since all changes to the application deployment are made by changing the files representing the application’s desired state, by observing the commits, pull request approvals, and merge requests, we can understand what changes have been made in the cluster.

For example, running the `git log` command on the sample-app-deployment repository displays all the commits made to this repo since it was created.

```
$ git log --no-merges
commit ce920168912a7f3a6cdd57d47e630ac09aebc4e1 (origin/tekenstam-patch-2)
Author: Todd Ekenstam <tekenstam@gmail.com>
Date: Mon Nov 9 13:59:25 2020 -0800

    Reduce Replica count back to 1

commit 8613d1b14c75e32ae04f3b4c0470812e1bdec01c (origin/tekenstam-patch-1)
Author: Todd Ekenstam <tekenstam@gmail.com>
Date: Mon Nov 9 13:58:26 2020 -0800

    Update Replica count to 3

commit 09d6663dcfa0f39b1a47c66a88f0225a1c3380bc
Author: tekenstam <tekenstam@gmail.com>
Date: Wed Feb 5 22:14:35 2020 -0800

    Update deployment.yaml

commit 461ac41630bfa3eee40a8d01dbcd2a5cd032b8f1
Author: Todd Ekenstam <Todd_Ekenstam@intuit.com>
Date: Wed Feb 5 21:51:03 2020 -0800

    Update sample-app image to gitopsbook/sample-app:cc52a36

commit 99bb7e779d960f23d5d941d94a7c4c4a6047bb22
Author: Alexander Matyushentsev <amatyushentsev@gmail.com>
```

```
Date: Sun Jan 26 22:01:20 2020 -0800
```

```
Initial commit
```

From this output, we can see Alex authored the first commit to this repo on Jan 26. The most recent commit was authored by Todd which, according to the commit title, reduces the Replica count back to one. We can look at the actual differences in the commit by running the following command.

```
$ git show ce920168912a7f3a6cdd57d47e630ac09aebc4e1
commit ce920168912a7f3a6cdd57d47e630ac09aebc4e1 (origin/tekenstam-patch-2)
Author: Todd Ekenstam <tekenstam@gmail.com>
Date: Mon Nov 9 13:59:25 2020 -0800

    Reduce Replica count back to 1

diff --git a/deployment.yaml b/deployment.yaml
index ed2398a..5fc3833 100644
--- a/deployment.yaml
+++ b/deployment.yaml
@@ -3,7 +3,7 @@ kind: Deployment
  metadata:
    name: sample-app
  spec:
-  replicas: 3
+  replicas: 1
    revisionHistoryLimit: 3
  selector:
    matchLabels:
```

From this we see that the line `replicas: 3` was changed to `replicas: 1`.

The same information is available in the GitHub UI.

The screenshot shows the GitHub commit history for the repository `gitopsbook / sample-app-deployment` on the `master` branch. The commits are organized into three groups by date:

- Commits on Nov 9, 2020:**
 - Merge pull request #8 from `gitopsbook/tekenstam-patch-2` (Verified) - `a301fbe`
 - Reduce Replica count back to 1 (Verified) - `ce92816`
 - Merge pull request #7 from `gitopsbook/tekenstam-patch-1` (Verified) - `39936ce`
 - Update Replica count to 3 (Verified) - `8613d1b`
- Commits on Feb 5, 2020:**
 - Update deployment.yaml (Verified) - `89d6663`
 - Update sample-app image to `gitopsbook/sample-app:cc52a36` (Verified) - `461ac41`
- Commits on Jan 26, 2020:**
 - Initial commit (Verified) - `99bb7e7`

At the bottom of the commit list are "Newer" and "Older" buttons.

Figure 8.13 Reviewing the GitHub commit history of a deployment repository allows you to see all the changes that have been made to the application deployment over time.

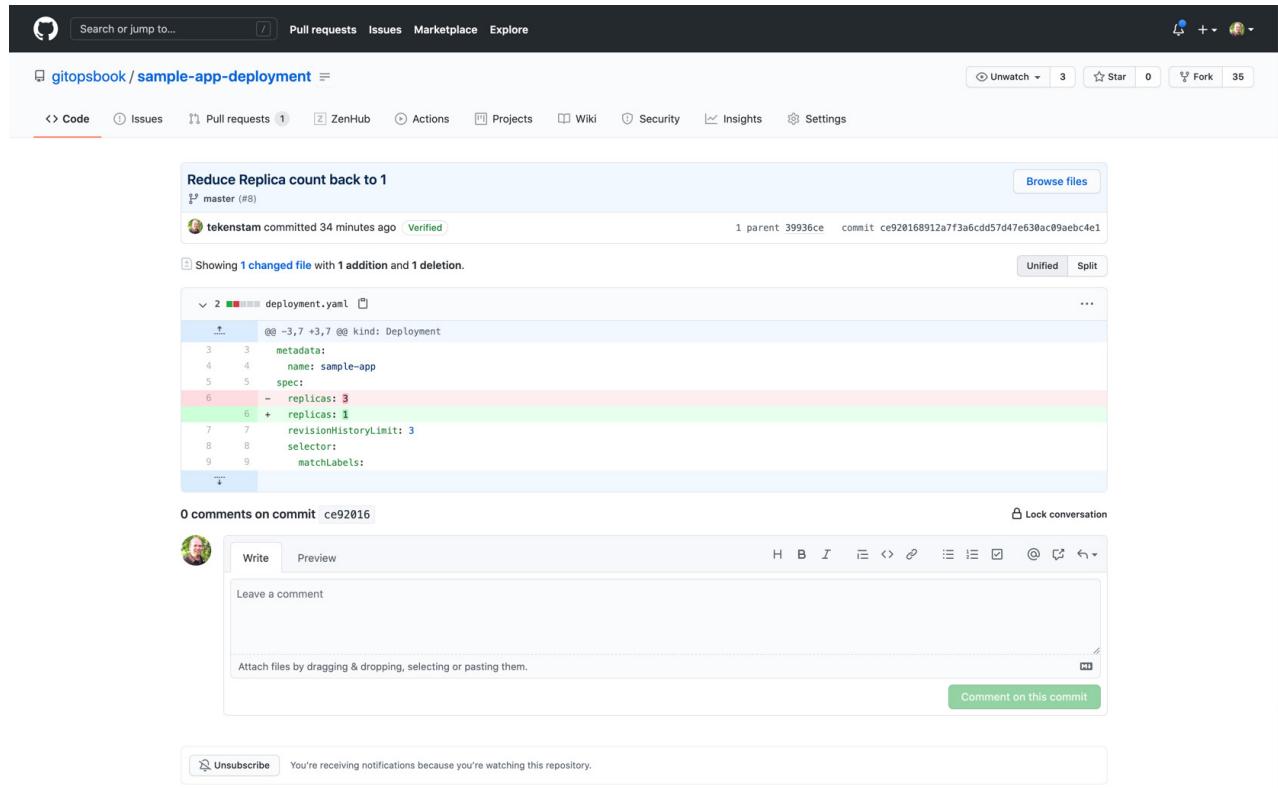


Figure 8.14 The detail of individual commits can be inspected.

Having a deployment audit log for each of your applications deployed to your cluster is critical for managing them at scale. If only one person changes a cluster, if something breaks, likely that person will know what change they may have made that caused it to break. But if you have multiple teams, perhaps in different geographic locations and time zones, it's imperative to be able to answer the question "Who deployed the app last, and what changes did they make?"

As we've learned in previous chapters, the GitOps repository is a collection of changes, additions, and deletions to the repository files representing the desired state of the system. Each modification to the repository is known as a "commit." Just as application logging helps provide the history of what occurred in the code, Git provides a log that provides the history of changes to the repository. We can examine the Git log to "observe" the changes that have occurred to the repository over time.

For GitOps, looking at the deployment repository logs is just as important as looking at the application logs when it comes to the observability of the GitOps system. Because the GitOps repository is the source of truth for the desired state of the system, examining the logs of the repository shows us when and why (if the commit comment is descriptive

enough) changes were made to the desired state of the system, as well as who made and approved those changes.

This is a critically important aspect of how GitOps provides observability to users. While the GitOps operator or service may also provide application logs detailing its execution, the Git log of the deployment repository usually will give you an excellent understanding of what changes have been going on in the system.

Exercise 8.11

Using the same Git fork of the sample-app-deployment repository you've been using throughout this chapter, run the command `git log`. Examine the output. Can you trace your process through the sections of this chapter? Do you see any earlier commits to this repo by the authors?

8.4 Summary

- Aspects of Observability can be measured by monitoring Event Logging, Metrics, and Tracing.
- A data collector such as Logstash, fluentd, or Scribe can collect the application output (events) in stdout and store the log messages in a centralized data store for later analysis.
- Observe the application output using `kubectl logs`
- Prometheus collects metrics from both nodes and pods to provide a snapshot of the performance and operation of all system components.
- Use Jaeger (Open Tracing) to monitor distributed calls to gain system insight such as error and latency
- Application Health: Is the application operating correctly? If a new version of the application is deploying using GitOps, is the system "better" than before?
- Use `kubectl describe` to monitor application health.
- Application Sync Status: Is the running state of the application the same as the desired state defined in the deployment Git repo?
- Configuration Drift: Has the application's configuration been changed outside of the declarative GitOps system (e.g., either manually or imperatively)?
- Use `kubectl diff` and `kubediff` to detect Application Sync Status and Configuration Drift
- GitOps Change Log: What changes were recently made to the GitOps system? Who made them, and for what reason?

9

Argo CD

This chapter covers:

- What is Argo CD?
- Deploying an application using Argo CD
- Using Argo CD Enterprise Features

In this chapter you will learn how to use the Argo CD GitOps operator to deploy our reference example application to Kubernetes. You will also learn how Argo CD addresses enterprise considerations such as single sign-on (SSO) and access control.

We recommend you read chapters 1, 2, 3, and 5 before reading this chapter.

9.1 What is Argo CD?

Argo CD is an open-source GitOps operator for Kubernetes⁸⁸. The project is a part of the Argo family - a set of cloud-native tools for running and managing jobs and applications on Kubernetes. Along with Argo Workflows, Rollouts, and Events, Argo CD focuses on application delivery use cases and makes it easier to combine three modes of computing: services, workflows, and event-based processing. In 2020 the Argo was accepted by the Cloud Native Computing Foundation (CNCF) as an incubation-level hosted project.

NOTE CNCFLinux Foundation project that hosts critical components of the global technology infrastructure.

The company behind Argo CD is Intuit - the creator of TurboTax and QuickBooks. In early 2018 Intuit decided to adopt Kubernetes to speed up cloud migration. At the time, the market already had several successful continuous deployment tools for Kubernetes, but none of them fully satisfied Intuit's needs. So instead of adopting an existing solution, the

⁸⁸ <https://argoproj.github.io/projects/argo-cd>

company decided to invest in a new project and started working on Argo CD. So what is so special about Intuit's requirements? The answer to that question explains how Argo CD is different from other Kubernetes CD tools and explains the main project use cases.

9.1.1 Main use cases

The importance of a GitOps methodology and benefits of representing infrastructure as code is not questionable. However, the enterprise-scale applies additional requirements. Intuit is a cloud-based software-as-a-service company. With around five thousand developers, the company successfully runs hundreds of micro-services on-premise and in the cloud. Given that scale, it was unreasonable to expect that every team would run its own Kubernetes cluster. Instead, it was decided that a centralized platform team would run and maintain a set of multi-tenant clusters for the whole company. At the same time, the end-users should have the freedom and necessary tools to manage workloads in those clusters. These considerations have defined the following additional requirements on top of the decision to use GitOps:

AVAILABLE AS A SERVICE

The simple onboarding process is extremely important if you are trying to move hundreds of micro-services to Kubernetes. Instead of asking every team to install, configure, and maintain the deployment operator, it should be provided by the centralized team. With several thousands of new users, SSO integration is crucial. The service must integrate with various SSO providers instead of introducing its own user management.

ENABLE MULTI-TENANCY AND MULTI-CLUSTER MANAGEMENT

In multi-tenant environments, users need an effective and flexible access control system. Kubernetes has a great built-in role-based access control system, but that is not enough when you have to deal with hundreds of clusters. The continuous deployment tool should provide access control on top of multiple clusters and seamlessly integrate with existing SSO providers.

ENABLE OBSERVABILITY

Last, but not least, the continuous deployment tool should provide developers insights about the state of managed applications. That assumes the user-friendly interface that quickly answers the following questions:

- Is the application configuration in sync with the configuration defined in Git?
- What exactly is not matching?
- Is the application up and running?
- What exactly is broken?

The company needed the GitOps operator ready for enterprise scale. The team evaluated several GitOps operators, but none of them satisfied all requirements, so it was decided to implement Argo CD.

Exercise 9.1

Reflect on your organization's needs and compare it to use cases that Argo CD is focused on. Try to answer if Argo CD solves the pain points your team has.

9.1.2 Core Concepts

In order to effectively use Argo CD, we should understand two basic concepts: the Application and the Project. Let's have a closer look at the application first.

APPLICATION

The Application - provides a logical grouping of Kubernetes resources and defines the resources manifests source and destination.

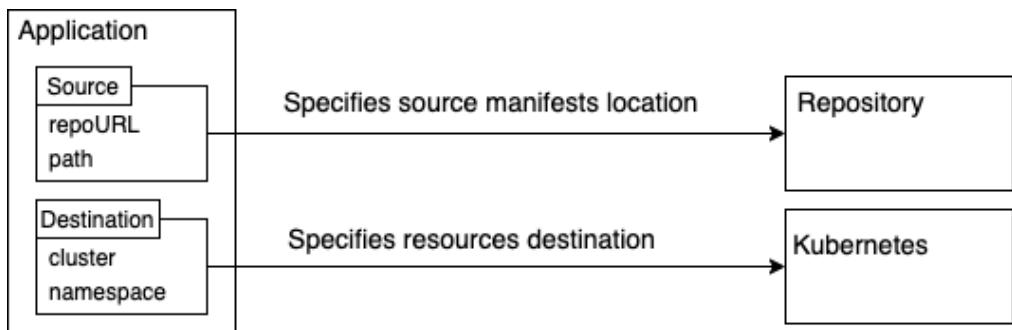


Figure 9.1 The main properties of the Argo CD Application are source and destination. The source specifies resource manifests location in the Git repository. The destination specifies where resources should be created in the Kubernetes cluster.

The Application source - includes the repository URL and the directory inside of the repository. Typically repositories include multiple directories, one per application environments such as QA and Production. The sample directory structure of such a repository is represented below:

```

.
├── prod
│   └── deployment.yaml
└── qa
    └── deployment.yaml
  
```

Each directory does not necessarily contain plain YAML files. Argo CD does not enforce any configuration management tool and instead provides first-class support for various config management tools. So the directory might as well contain a Helm chart definition or YAML along with Kustomize overlays.

The Application destination defines where resources must be deployed and includes the API server URL of the target Kubernetes cluster, along with the cluster namespace name. The API server URL identifies the cluster where all application manifests must be deployed. It is impossible to deploy application manifests across several clusters, but different

applications might be deployed into different clusters. The namespace name is used to identify the target namespace of all namespace level application resources.

So the Argo CD application represents an environment deployed in the Kubernetes cluster and connects it to the desired state stored in the Git repository.

Exercise 9.2

Consider the real service deployed in your organization and come up with a list of Argo CD applications. Define the source repository URL, directory, and target cluster with the namespace for one of the applications from your list.

In addition to the source and destination, the Argo CD application has another two important properties: Sync and Health statuses.

Sync Status answers whether the observed application resources state deviates from the resources state stored in the Git repository. The logic behind deviation calculation is equivalent to the logic of the `kubectl diff` command. The possible values of a sync status are In-Sync and Out-Of-Sync. The In-Sync status means that each application resource is found and fully matching to the expected resource state. The Out-Of-Sync status means that at least one resource status is not matching to the expected state or not found in the target cluster.

The health status aggregates information about the observed health status of each resource that comprises the application. The health assessment logic is different for each Kubernetes resource type and resulted in one of the following values:

- **Healthy** - for example, the Kubernetes deployment is considered Healthy if the required number of Pods is running and each Pod successfully passing both readiness and liveness probes
- **Progressing** - represents a resource that is not healthy yet but still expected to reach a healthy state. The Deployment is considered progressing if it is not healthy yet but still without time limit specified by the `progressingDeadlineSeconds` ⁶⁷field.
- **Degraded** - the antipode of Healthy status. The example is a Deployment that could not reach a healthy status within an expected timeout.
- **Missing** - represents the resource that is stored in Git but not deployed to the target cluster.

The aggregated application status is the worst status of every application resource. The Healthy status is the best, then goes Progressing, Degraded, and the Missing is the worst. So if all application resources are Health and only one is Degraded, then aggregated status is also Degraded.

Exercise 9.3

Consider an application consisting of two Deployments. The following information is known about the resources:

- Deployment one has an image that does not match the image stored in the Git repository. All Deployment Pods have failed to start for several hours while Deployment `progressingDeadlineSeconds` is set to 10 minutes.

⁶⁷ <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds>

- The Deployment two is not fully matching the expected state and has all Pods running.

What are the application Sync and Health statuses?

The Health and Sync statuses answers the two most important questions about an application:

- Is my application working?
- Am I running what is expected?

PROJECT

Argo CD applications provide a very flexible way to manage different applications independently from each other. This functionality provides very useful insights to the team about each piece of infrastructure and greatly improves productivity. However, this is not enough to support multiple teams with different access levels:

- The mixed list of applications creates confusion that creates a human error possibility.
- Different teams have different access levels. As has been described in Chapter 6, the individual might use the GitOps operator to escalate his own permissions to get full cluster access.

The workaround for these issues is a separate Argo CD instance for each team. This is not a perfect solution since a separate instance means management overhead. In order to avoid management overhead, Argo CD introduces the Project abstraction. Figure 9.2 illustrates the relationship between applications and projects.

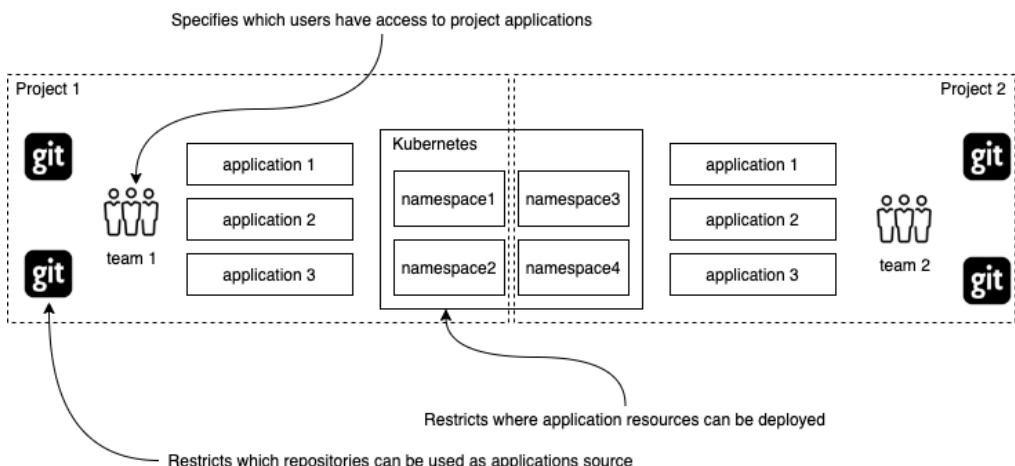


Figure 9.2 Demonstrates the relationship between applications and projects. The project provides a logical grouping of applications, isolates teams from each other that enables using Argo CD in multi-tenant environments.

The Project - provides a logical grouping of applications, isolates teams from each other, and allows for fine-tuning access control in each Project.

In addition to the separating sets of applications, the Project provides the following set of features:

- Restricts which Kubernetes clusters and Git repositories might be used by project applications.
- Restricts which Kubernetes resources can be deployed by each application within the project.

Exercise 9.4

Try to come up with a list of projects in your organization. Using projects, you can restrict what kind of resource users can deploy, source repositories, and destination clusters available within the project. Which restrictions would you configure for your projects?

9.1.3 Architecture

At first glance, the implementation of the GitOps operator does not look too complex. In theory, all you need is to clone the Git repository with manifests and use kubectl diff and kubectl apply to detect and handle config drifts. This is true until you are trying to automate this process for multiple teams and managing the configuration of dozens of clusters simultaneously. Logically this process is split into three phases, and each phase has its own challenges:

- Retrieve resource manifests.
- Detect and fix the deviations.
- Present the results to end-users.

Each phase consumes different resources, and the implementation of each phase has to scale differently. A separate Argo CD component is responsible for each phase.

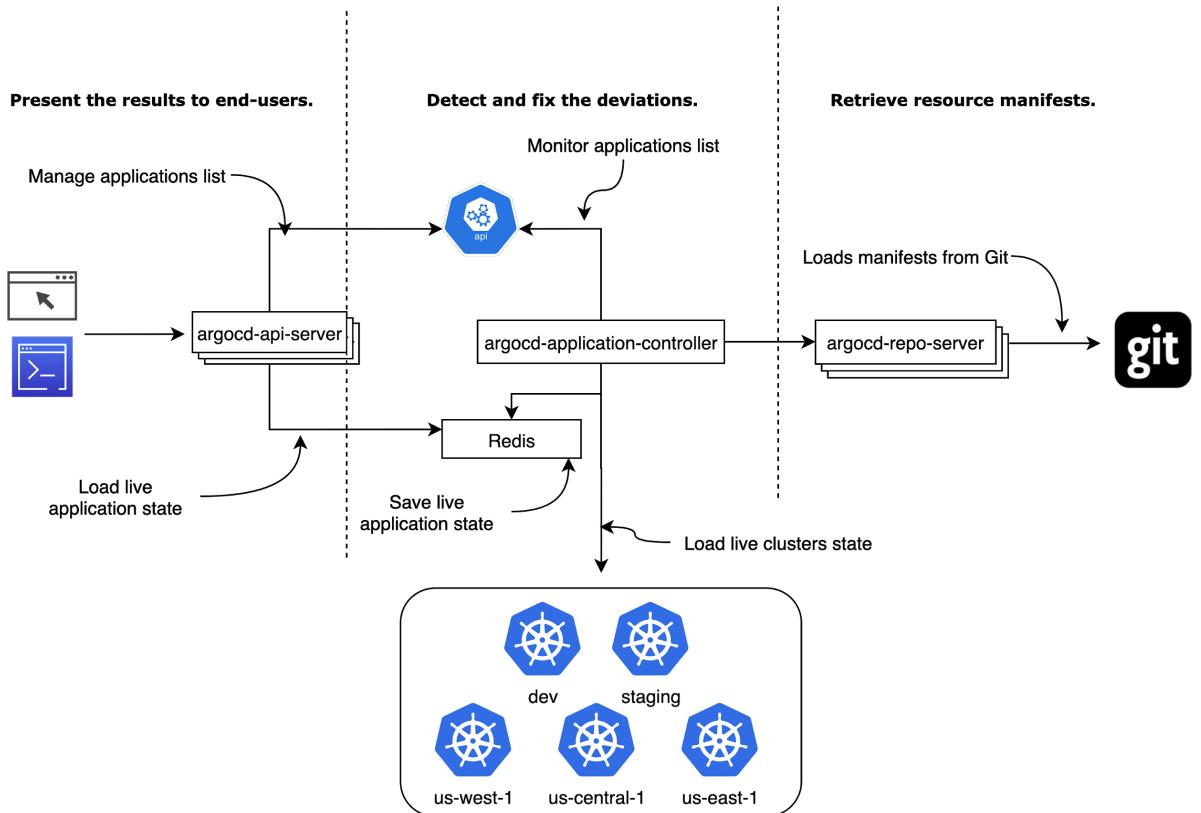


Figure 9.3 Argo CD consists of three main components that implement GitOps reconciliation cycle phases. The **argocd-repo-server** retrieves manifests from Git. The **argocd-application-controller** compares manifests from Git with resources in Kubernetes cluster. The **argocd-api-server** presents reconciliation results to the user.

Let's go through each phase and the corresponding Argo CD component implementation details.

RETRIEVE RESOURCE MANIFESTS

The manifest generation in Argo CD is implemented by the “**argocd-repo-server**” component. This phase presents a whole set of challenges.

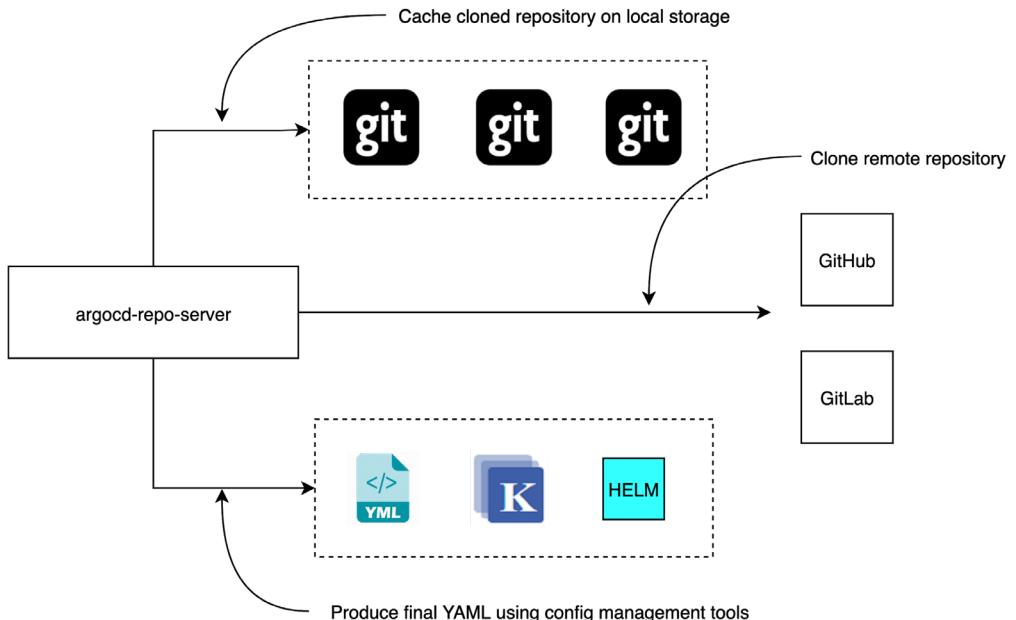


Figure 9.4 argocd-repo-server caches cloned repository on local storage and encapsulates interaction with the config management tool that is required to produce final resource manifests.

The manifests generation requires to download Git repository content and produce ready to use manifests YAML. First of all, it is too time-consuming to download the whole repository content every time when you need to retrieve expected resource manifests. Argo CD solves it by caching the repository content on local disk and using git fetch command to download only recent changes from the remote Git repository. The next challenge is related to memory usage. In real life, resources manifest are rarely stored as plain YAML files. In most cases, developers prefer to use a config management tool such as Helm or Kustomize. Every tool invocation causes a spike in memory usage. To handle the memory usage issues, Argo CD allows the user to limit the number of parallel manifest generations and scale up the number of "argocd-repo-server" instances to improve performance.

Detect and fix the deviations

The reconciliation phase is implemented by the "argocd-application-controller" component. The controller loads the live Kubernetes cluster state, compares it with the expected manifests provided by the "argocd-repo-server", and patches deviated resources. This phase is probably the most challenging one. In order to correctly detect deviations, the GitOps operator needs to know about each resource in the cluster, compare and update thousands of resources.

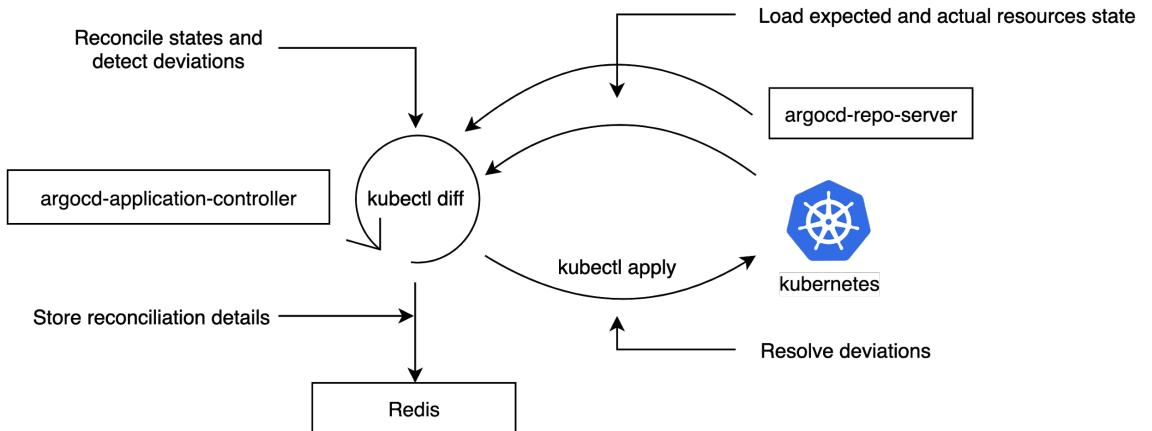


Figure 9.5 argocd-application-controller performs resources reconciliation. The controller leverages argocd-repo-server component to retrieve expected manifests and compare manifests with the light-weight in-memory Kubernetes cluster state cache.

The controller maintains a lightweight cache of each managed cluster and updates it in the background using Kubernetes watch API. This allows the controller to perform reconciliation on an application within a fraction of a second and empowers it to scale and manage dozens of clusters simultaneously. After each reconciliation, the controller has the exhausting information about each application resource, including the sync and health status. The controller saves that information into the Redis cluster so it could be presented to the end-user later.

PRESENT THE RESULTS TO END-USERS

Finally, the reconciliation results must be presented to end-users. This task is performed by the "argocd-server" component. While the heavy lifting is already done by the "argocd-repo-server" and "argocd-application-controller", this last phase has the highest resiliency requirements. The "argocd-server" is a stateless web application that loads the information about reconciliation results and powers the web user interface.

The architecture design allows Argo CD to serve GitOps operations a whole company and minimize the maintenance overhead.

Exercise 9.5

Which components serve user requests and require multiple replicas for resiliency?

Which components might require a lot of memory to scale?

9.2 Deploy your first application

While Argo CD is an enterprise-ready, complex distributed system, it is still lightweight and can easily run on minikube. The installation is trivial and includes a few simple steps. Please

refer to Appendix B for more information on how to install Argo CD or follow the official Argo CD instructions⁶⁸.

9.2.1 Deploying the first application

As soon as Argo CD is running, we are ready to deploy our first application. As it's been mentioned before, to deploy an Argo CD application, we need to specify the Git repository that contains deployment manifests and target Kubernetes cluster and namespace. To create the Git repository for this exercise, open the following Github repository and create a repository fork⁶⁹.

```
https://github.com/gitopsbook/sample-app-deployment
```

Argo CD can deploy into the external cluster as well as into the same cluster where it is installed. Let's use the second option and deploy our application into the default namespace of our Minikube cluster.

NOTE *Reset your fork* Have you already forked the deployment repository while working on previous chapters? Please make sure to revert changes for the best experience. The simplest way is to delete the previously forked repository and fork it again.

The application might be created using the Web user interface, using the CLI, or even programmatically using the REST or gRPC APIs. Since we already have Argo CD CLI installed and configured, let's use it to deploy an application. Please go ahead and execute the following command to create an application:

```
$ argocd app create sample-app \#A
  --repo https://github.com/<username>/sample-app-deployment \#B
  --path . \#C
  --dest-server https://kubernetes.default.svc \#D
  --dest-namespace default \#E
```

#A Unique application name.

#B Git repository URL.

#C Directory path within the Git repository.

#D The Kubernetes API server URL. The <https://kubernetes.default.svc/> is the API server URL that is available inside of every Kubernetes cluster.

#E The Kubernetes namespace name.

As soon as the application is created, we can use the Argo CD CLI to get the information about the application state. Use the following command to get the information about the "sample-app" application state:

```
argocd app get sample-app \#A
Name:           sample-app
Project:        default
Server:         https://kubernetes.default.svc
Namespace:      default
URL:           https://<host>:<port>/applications/sample-app
```

⁶⁸ https://argoproj.github.io/argo-cd/getting_started/

⁶⁹ <https://help.github.com/en/github/getting-started-with-github/fork-a-repo>

```

Repo: https://github.com/<username>/sample-app-deployment
Target:
Path:
SyncWindow: .
Sync Policy: Sync Allowed
Sync Status: <none>
Sync Status: OutOfSync from (09d6663) #B
Health Status: Missing #C

GROUP KIND NAMESPACE NAME STATUS HEALTH HOOK MESSAGE
Service default sample-app OutOfSync Missing
apps Deployment default sample-app OutOfSync Missing

```

#A CLI command that returns the information about an application state.

#B Application sync status that answers whether the application state matches the expected state or not.

#C Application aggregated health status.

As we can see from the command output, the application is out-of-sync and not healthy. By default, Argo CD does not push resources defined in the Git repository into the cluster even if it detects the deviation. In addition to the high-level summary, we can see the details of every application resource. Argo CD detected that the application is supposed to have a Deployment and a Service, but both resources are missing. To deploy the resources, we need to either configure automated application syncing using the sync policy⁷⁰ or trigger syncing manually. To trigger the sync and deploy the resources use the following command:

```

$ argocd app sync sample-app
TIMESTAMP GROUP KIND NAMESPACE #A NAME STATUS
    HEALTH HOOK MESSAGE #B
2020-03-17T23:16:50-07:00 Service default sample-app OutOfSync
    Missing
2020-03-17T23:16:50-07:00 apps Deployment default sample-app OutOfSync
    Missing

Name: sample-app
Project: default
Server: https://kubernetes.default.svc
Namespace: default
URL: https://<host>:<port>/applications/sample-app
Repo: https://github.com/<username>/sample-app-deployment
Target:
Path:
SyncWindow: .
Sync Policy: Sync Allowed
Sync Status: <none>
Sync Status: OutOfSync from (09d6663)
Health Status: Missing

Operation: Sync
Sync Revision: 09d6663dcfa0f39b1a47c66a88f0225a1c3380bc
Phase: Succeeded
Start: 2020-03-17 23:17:12 -0700 PDT
Finished: 2020-03-17 23:17:21 -0700 PDT
Duration: 9s
Message: successfully synced (all tasks run)

GROUP KIND NAMESPACE NAME STATUS HEALTH HOOK MESSAGE
#C

```

⁷⁰ https://argoproj.github.io/argo-cd/user-guide/auto_sync/

apps	Service created	default	sample-app	Synced	Healthy	service/sample-app
	Deployment created	default	sample-app	Synced	Progressing	deployment.apps/sample-app

#A CLI command that triggers application sync

#B Initial application state before the sync operation

#C Final application state after the sync is completed

As soon as the sync is triggered, Argo CD will push the manifests stored in Git into the Kubernetes cluster and then re-evaluates the application state. The final application state is printed to the console when the synchronization completes. The “sample-app” application was successfully synced, and each result matches the expected state.

9.2.2 Inspect application using user interface

In addition to the CLI and API, Argo CD provides a user-friendly Web interface. Using the web interface, you might get the high-level view of all your applications deployed across multiple clusters as well as get very detailed information about every application resource. Open the <https://<host>:<port>> URL to see the applications list in the Argo CD user interface.

The screenshot shows the Argo CD user interface for managing applications. On the left, there's a sidebar with icons for Home, Applications, Workspaces, and Help. The main header says "Applications" and "v1.4.2+4". Below the header, there are buttons for "+ NEW APP" and "SYNC APPS". A search bar is followed by a "FILTER BY:" dropdown menu containing sections for "SYNC" and "HEALTH". The "SYNC" section has checkboxes for "Synced" (1), "Unknown" (0), and "OutOfSync" (0). The "HEALTH" section has checkboxes for "Healthy" (1), "Unknown" (0), "Progressing" (0), and "Suspended" (0). The main content area displays a table for the "sample-app" application. The table includes columns for Project (default), Labels, Status (Healthy, Synced), Repository (https://github.com/alexmt/sample-app...), Target Ref., Path, Destination (https://kubernetes.default.svc), and Namespace (default). At the bottom of the table are three buttons: "SYNC", "REFRESH", and "DELETE". Above the table, there are filters for "Application list filter" and "Applications tiles". On the right side of the page, there are "Page view settings" (grid, list, etc.) and a "Logout" button. The bottom right corner shows "page size: 10 ▾".

Figure 9.6 Applications list page visualizes available Argo CD applications. The page provides high-level information about each application, such as sync and health status.

The applications list page provides high-level information about all deployed applications, including health and synchronization status. Using this page, you can quickly find if any of your applications have degraded or have configuration drift. The user interface is designed for large enterprises and able to handle hundreds of applications. You can use search and various filters to quickly find the desired applications.

Exercise 9.6

Experiment with the filters and page view settings to learn which other features available in the applications list page.

The additional information about the application is available on the application details page. Navigate to the application details page by clicking on the "sample app" application tile.

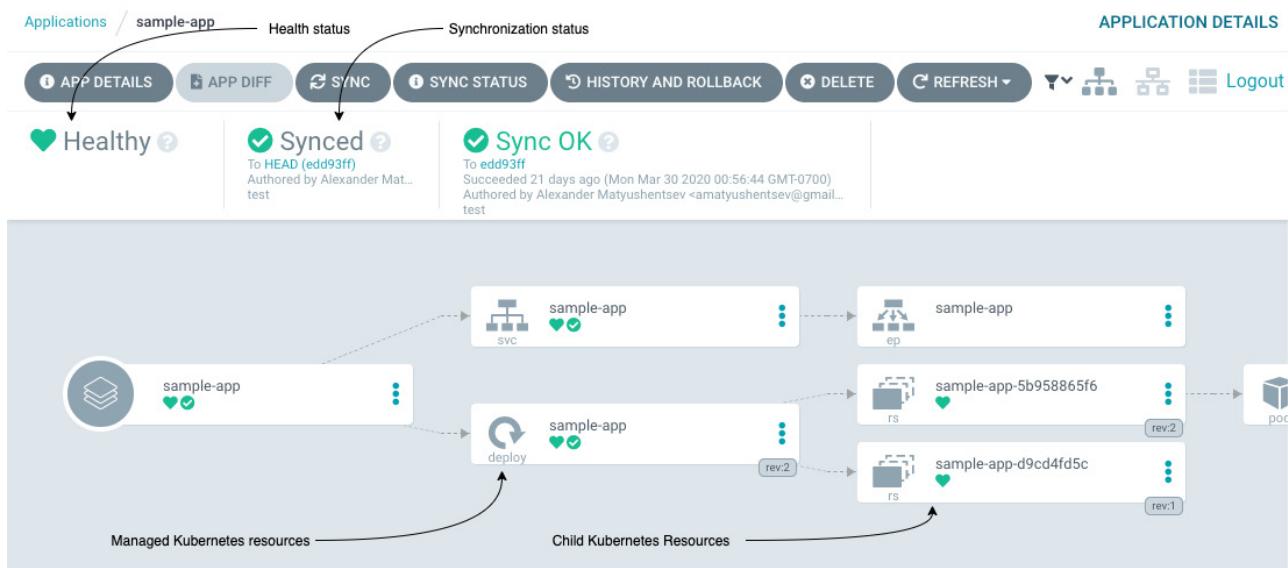


Figure 9.7 Application details page provides information about the application resource hierarchy as well as detailed information about each resource.

The application details page visualizes the application resources hierarchy and provides additional details about synchronization and health status. Let's take a look closer at the application resources tree and learn which features it provides.

The root element of the resource tree is the application itself. The next level consists of managed resources. The managed resources are resources that manifest defined in Git and controlled by Argo CD explicitly. As we've learned in chapter 2, the Kubernetes controllers often leverage delegation and create child resources to delegate the work. The third and deeper levels represent such resources. That provides complete information about every application element and makes the application details page an extremely powerful Kubernetes dashboard.

In addition to the information, the user interface allows executing various actions against each resource. It is possible to delete any resource, recreate it by running sync action, update the resource definition using a built-in YAML editor, and even run resource specific actions such as Deployment restart.

Exercise 9.7

Go ahead, use the Application details page to inspect your application. Try to find how to view the resource manifests; locate Pods, and find a way to see the live logs.

9.3 Deep dive into Argo CD features

So far, we've learned how to deploy new applications using Argo CD and get detailed application information using CLI and the user interface. Next, let's learn how to deploy a new application version using GitOps and Argo CD.

9.3.1 GitOps driven deployment

In order to perform GitOps deployment, we need to update resource manifests and let the GitOps operator push changes into the Kubernetes cluster. As a first step clone the deployment repository using the following command:

```
$ git clone git@github.com:<username>/sample-app-deployment.git
$ cd sample-app-deployment
```

Next, use the following command to change the image version of the Deployment resource.

```
$ sed -i '' 's/sample-app:v.*/sample-app:v0.2/' deployment.yaml
```

Use the `git diff` command to make sure that your Git repository has expected changes:

```
$ git diff
diff --git a/deployment.yaml b/deployment.yaml
index 5fc3833..397d058 100644
--- a/deployment.yaml
+++ b/deployment.yaml
@@ -16,7 +16,7 @@ spec:
  containers:
  - command:
    - /app/sample-app
-   image: gitopsbook/sample-app:v0.1
+   image: gitopsbook/sample-app:v0.2
     name: sample-app
   ports:
   - containerPort: 8080
```

Finally use `git commit` and `git push` to push changes to the remote Git repository:

```
$ git commit -am "update deployment image"
$ git push
```

Let's use the Argo CD CLI to make sure that Argo CD correctly detected manifest changes in Git and then trigger a synchronization process to push the changes into the Kubernetes cluster.

```
$ argo cd app diff sample-app --refresh
===== apps/Deployment default/sample-app =====
21c21
<       image: gitopsbook/sample-app:v0.1
---
```

```
>     image: gitopsbook/sample-app:v0.2
```

Exercise 9.8

Open Argo CD UI and use the application details page to check application sync status and inspect managed resources status.

Use the `argocd sync` command to trigger the synchronization process:

```
$ argocd app sync sample-app
```

Great, you just performed GitOps deployment using Argo CD!

9.3.2 Resource Hooks

Resource manifests syncing is just the basic use-case. In real life, we often need to execute additional steps before and after actual deployment. For example, set the maintenance page, execute database migration before the new version deployment and finally remove the maintenance page.

Traditionally these deployment steps are scripted in the CI pipeline. However, this again requires production access from the CI server that possesses a security threat. To solve that problem, Argo CD provides a feature called Resource Hooks. These hooks allow running custom scripts, typically packaged into a Pod or a Job, inside of the Kubernetes cluster during the synchronization process.

The hook is a Kubernetes resource manifest stored in the Git repository and annotated with the `argocd.argoproj.io/hook` annotation. The annotation value contains a comma-separated list of phases when the hook is supposed to be executed. Following phases are supported:

- **PreSync** - Executes prior to the applying of the manifests.
- **Sync** - Executes after all PreSync hooks completed and were successful, at the same time as the apply of the manifests.
- **Skip** - Indicates to Argo CD to skip the apply of the manifest.
- **PostSync** - Executes after all Sync hooks completed and were successful, a successful apply, and all resources in a Healthy state.
- **SyncFail** - Executes when the sync operation fails.
- The hooks are executed inside of the cluster, so there is no need to access the cluster from the CI pipeline. The ability to specify the sync phase provides the necessary flexibility and allows a mechanism to solve the majority of real-life deployment use cases.

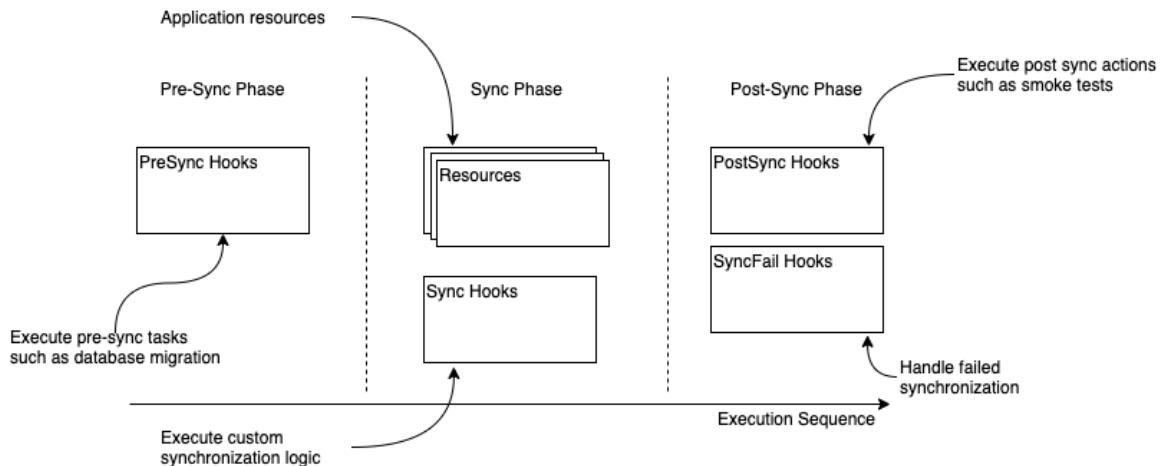


Figure 9.8 The synchronization process includes three main phases. The pre-sync phase is used to execute preparation tasks such as database migration. The sync phase includes the synchronization of application resources. Finally, the post-sync phase runs post-processing tasks, such as email notifications.

It is time to see the hooks feature in action! Add the hook definition into the sample app deployment repository and push changes to the remote repository:

```
$ git add pre-sync.yaml
$ git commit -am 'Add pre-sync hook'
$ git push
```

Listing 9.1 [pre-sync.yaml](#).

```
apiVersion: batch/v1
kind: Job
metadata:
  name: before
  annotations:
    argocd.argoproj.io/hook: PreSync
spec:
  template:
    spec:
      containers:
        - name: sleep
          image: alpine:latest
          command: ["echo", "pre-sync"]
          restartPolicy: Never
    backoffLimit: 0
```

Argo CD user interface provides much better visualization of a dynamic process than CLI. Let's use it to better understand how hooks work. Open the Argo CD UI using the following command:

```
$ minikube service argocd-server -n argocd --url
```

Navigate to the “sample-app” details page and trigger the synchronization process using the “Sync” button. The syncing process is represented in the figure below:

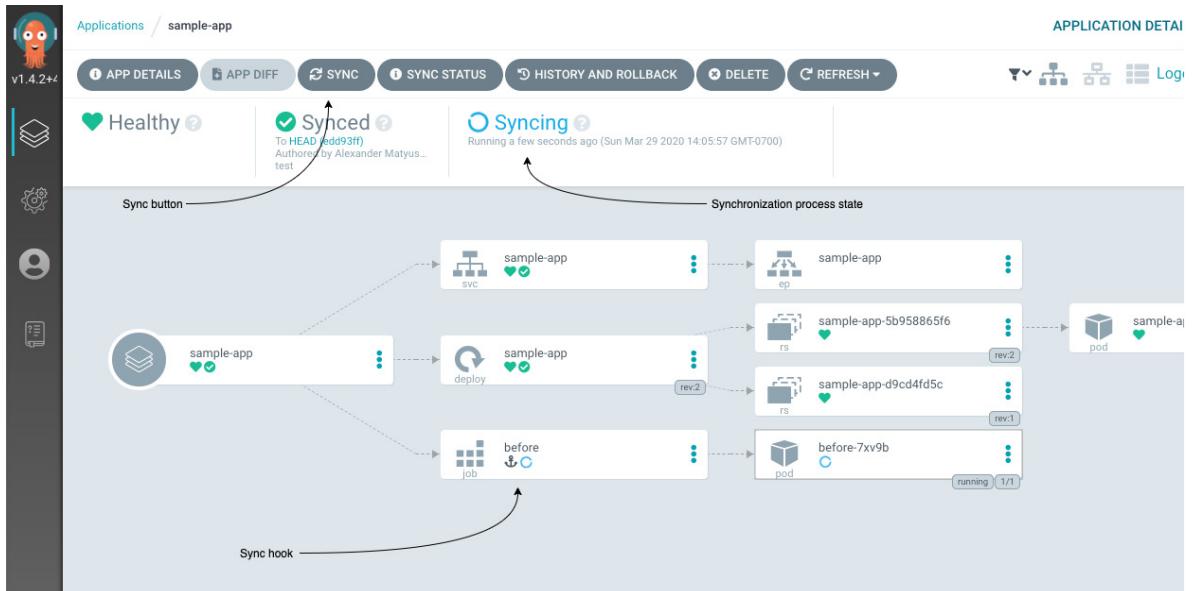


Figure 9.9 The application detail page allows the user to trigger the synchronization as well as view detailed information about the synchronization progress including synchronization hooks.

As soon as the sync is started, the application details page shows live process status in the top right corner. The status includes information about operation start time and duration. You can view the syncing status panel with detailed information, including sync hook results, by clicking the sync status icon.

The hooks are stored as the regular resource manifests in the Git repository and also visualized as regular resources in the Application resource tree. You can see the real-time status of the “before” job and use the Argo CD user interface to inspect child Pods.

In addition to phase, you might customize hook deletion policy. The deletion policy allows automating hook resources deletion that will save you a lot of manual work.

Exercise 9.9

Read more details in the Argo CD documentation⁷¹ and change the “before” Job deletion policy. Use the Argo CD user interface to observe how various deletion policies affect hook behavior. Synchronize application and observe how hook resources got created and deleted by Argo CD.

⁷¹ https://argoproj.github.io/argo-cd/user-guide/resource_hooks/#hook-deletion-policies

9.3.3 Post-deployment verification

Resource hooks allow encapsulating the application synchronization logic, so we don't have to use scripts and continuous integration tools. However, some of such use-cases naturally belong to continuous integration processes, and it is still preferable to use tools like Jenkins.

One of such use-cases is the post-deployment verification. The challenge here is that GitOps deployment is asynchronous by nature. After the commit is pushed to the Git repository, we still need to make sure that changes are propagated to the Kubernetes cluster. Even after changes are propagated, it is not safe to start running tests. In most cases, the update of a Kubernetes resource is not instant either. For example, the Deployment resource update triggers the rolling update process. The rolling update might take several minutes or even fail due if the new application version has an issue. So if you start tests too early, you might end up testing the previously deployed application version.

Argo CD makes this issue trivial by providing tools that help to monitor application status. The `argocd app wait` command monitors the application and exits after the application reaches a synched and healthy state. As soon as the command exits, you can assume that all changes are successfully rolled out, and it is safe to start post-deployment verification. The `argocd app wait` command is often used in conjunction with `argocd app sync`. Use the following command to synchronize your application and wait until the change is fully rolled out, and the application is ready for testing:

```
$ argocd app sync sample-app && argocd app wait sample-app
```

9.4 Enterprise features

Argo CD is pretty lightweight, and it is really easy to start using it. At the same time, it scales well for a large enterprise and able to accommodate the needs of multiple teams. The enterprise features can be configured as you go. If you are rolling out an Argo CD for your organization, then the first question is how to configure the end-user and effectively manage access control.

9.4.1 Single sign-on

Instead of introducing its own user management system, Argo CD provides integration with multiple single sign-on services (SSO). The list includes Okta, Google OAuth, Azure AD, and many more.

NOTE SSO SSO is a session and user authentication service that allows a user to use one set of login credentials to access multiple applications.

The SSO integration is great because it saves you a lot of management overhead, as well as for end-users, who don't have to remember another set of login credentials. There are several open standards for exchanging authentication and authorization data. The most popular ones are SAML, OAuth, and OpenID Connect (OIDC). Among those three, SAML and OIDC satisfy the best requirements of a typical enterprise and can be used to implement SSO. Argo CD decided to go ahead with OIDC because of its power and simplicity.

The number of steps required to configure an OIDC integration depends on your OIDC provider. The Argo CD community already contributed a number of instructions for popular OIDC providers such as Okta and Azure AD. After performing the configuration on the OIDC provider side, you need to add the corresponding configuration to the `argocd-cm` ConfigMap. The snippet below represents the sample Okta configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  url: https://<myargocdhost> #A
  oidc.config: | #B
    name: Okta
    issuer: https://yourorganization.oktapreview.com
    clientID: <your client id>
    clientSecret: <your client secret>
    requestedScopes: ["openid", "profile", "email", "groups"]
    requestedIDTokenClaims: {"groups": {"essential": true}}
```

#A The externally facing base URL Argo CD URL.

#B OIDC configuration that includes Okta application client id and secret

What if your organization does not have an OIDC compatible SSO service? In this case, you can use a federated OIDC provider Dex⁷² which is bundled into the Argo CD by default. Dex acts as a proxy to other identity providers and allows establishing integration with SAML, LDAP providers, or even services like GitHub and Active Directory.

GitHub often is a very attractive option, especially if it is already used by developers in your organization. Additionally, organizations and teams configured in GitHub naturally fit the access control model required to organize cluster access. As you are going to learn soon, it is very easy to model Argo CD access using the GitHub team membership. Let's use GitHub to enhance our Argo CD installation and enable SSO integration.

First of all, we need to create a GitHub OAuth application. Navigate to the <https://github.com/settings/applications/new> URL and configure the application settings as represented in the figure below:

⁷² <https://github.com/dexidp/dex>

Register a new OAuth application

Application name *



Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our [OAuth documentation](#) for more information.

Figure 9.10 New Github OAuth application settings include the application name and description, home page URL, and most importantly authorization callback URL.

Specify the application name of your choice and the home page URL that matches the Argo CD web user interface URL. The most important application setting is the callback URL. The callback URL value is the Argo CD web user interface URL plus the /api/dex/callback path. The sample URL with minikube might be <http://192.168.64.2:32638/api/dex/callback>.

After creating the application, you will be redirected to the OAuth application settings page. Copy the application Client ID and Client Secret. These values will be used to configure the Argo CD settings.

ArgoCD

You can list your application in the [GitHub Marketplace](#) so that other users can discover it.

[List this application in the Marketplace](#)

0 users

Client ID
acadd055b20ee1fa1c3d

Client Secret
fb132148f7258fd2679b74a2bfbc5ae2f1c7bbbd

[Revoke all user tokens](#) [Reset client secret](#)

Figure 9.11 Github OAuth application settings page displays the Client ID and Client Secret value which are required to configure the SSO integration.

Substitute the placeholder values in the `argocd-cm.yaml` file with your environment values:

Listing 9.2 `argocd-cm.yaml`.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  url: https://<minikube-host>:<minikube-port> #A
  dex.config: |
    connectors:
      - type: github
        id: github
        name: GitHub
        config:
          clientID: <client-id> #B
          clientSecret: <client-secret> #C
          loadAllGroups: true
```

#A The externally facing base URL Argo CD URL.

#B Github OAuth application client ID

#C Github OAuth application client secret

#C Your Github username

Update the Argo CD ConfigMap using the kubectl apply command:

```
$ kubectl apply -f ./argocd-cm.yaml -n argocd
```

You are ready to go! Open the Argo CD user interface in the browser and use the “LOGIN VIA GITHUB” button.

9.4.2 Access control

You might notice that after a successful login using Github SSO integration, the application list page is empty. If you try creating a new application, you will see a “permission denied” error. This behavior is expected because we have not given any permission to the new SSO user yet. In order to provide the user with appropriate access, we need to update Argo CD access control settings.

Argo CD provides a flexible role-based access control (RBAC) system which implementation is based on Casbin⁷³ - powerful open-source access control library. Casbin provides a very solid foundation and allows configuring various access control rules.

The RBAC Argo CD settings are configured using `argocd-rbac-cm` ConfigMap. To quickly dive into the configuration details, let’s update the ConfigMap fields and then go together through each change.

Substitute the `<username>` placeholder with your Github account username in the `argocd-rbac-cm.yaml` file:

Listing 9.3 `argocd-rbac-cm.yaml`.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-rbac-cm
  labels:
    app.kubernetes.io/name: argocd-rbac-cm
    app.kubernetes.io/part-of: argocd
data:
  policy.csv: |
    p, role:developer, applications, *, /*, allow
    g, role:developer, role:readonly

    g, <username>, role:developer
  scopes: '[groups, preferred_username]' #B
```

#A The `policy.csv` contains role-based access rules

#B The `scopes` setting specified which JWT claims used to infer user groups

Apply the RBAC changes using the kubectl apply command:

```
$ kubectl apply -f ./argocd-rbac-cm.yaml -n argocd
```

The `policy.csv` field in the configuration above defines a role named `role:developer` with full permissions on Argo CD applications and read-only permissions over Argo CD system

⁷³ <https://github.com/casbin/casbin>

settings. The role is granted to any user that belongs to a group whose name matches your Github account username. As soon as changes are applied, please refresh the Applications list page and try syncing the “sample-app” application.

We've introduced quite a few new terms. Let's step back and discuss what roles, groups, claims are and how they work together.

ROLE

The role allows or denies a set of actions on an Argo CD object to a particular subject. The role is defined in the following form:

```
p, subject, resource, action, object, effect
```

where:

- **p** - indicates the RBAC policy line
- **subject** - is a group
- **resource** - one of Argo CD resource types. Argo CD supports the following resources: “clusters”, “projects”, “applications”, “repositories”, “certificates”, “accounts”.
- **action** - an action name that might be executed against a resource. All Argo CD resources support the following actions: “get”, “create”, “update”, “delete”. The “*” value matches any action.
- **object** - a pattern that identifies a particular resource instance. The “*” value matches any instance.
- **effect** - defines whether the role grants or denies the action
- The `role:developer` role from the example above allows any action against any Argo CD application:

```
p, role:developer, applications, *, /*, allow
```

GROUP

Group provides the ability to identify a set of users and works in conjunction with OIDC integration. After performing the successful OIDC authentication, the end-user receives a JWT token that verifies the user identity as well as provides additional metadata stored in the token claims.

NOTE *JWT Token* A JWT Token is an internet standard for creating JSON-based access tokens that assert some number of claims.⁷⁴

The token is supplied with every Argo CD request. The Argo CD extract the list of groups that user belongs to from a configured list of token claims and use it to verify user permissions.

Take a look at the real like token claims example generated by Dex:

```
{
  "iss": "https://192.168.64.2:32638/api/dex",
  "sub": "CgY0MjY0MzcSBmdpdGh1Yg",
  "aud": "argo-cd",
```

⁷⁴ https://en.wikipedia.org/wiki/JSON_Web_Token

```

"exp": 1585646367,
"iat": 1585559967,
"at_hash": "rAz6dDHS1BWvU6PiWj_o9g",
"email": "AMatyushentsev@gmail.com",
"email_verified": true,
"groups": [
    "gitopsbook"
],
"name": "Alexander Matyushentsev",
"preferred_username": "alexmt"
}

```

The token contains two useful claims which might be useful for authorization:

- **groups** - includes a list of Github orgs and teams the user belongs to
- **preferred_username** - the Github account username

By default, Argo CD uses “groups” to retrieve list user groups from the JWT token. We’ve added the “preferred_username” claim using the “scopes” setting to allow identifying Github users by name.

Exercise 9.10

Update the `argocd-rbac-cm` ConfigMap to provide admin access to the Github user based on its email.

NOTE This chapter covers important foundations of Argo CD and gets you ready for further learning. Explore Argo CD documentation to learn about diffing logic customization, config management tools fine-tuning, advanced security features, such as auth tokens, and many more. The project keeps evolving and getting new features in every release. Checkout the Argo CD blog to stay up to date with the changes, and don’t hesitate to ask questions in Argoproj slack channel.

9.4.3 Declarative Management

As you might’ve noticed Argo CD provides a lot of configuration settings. The RBAC policies, SSO settings, Applications, and Projects - all of that are settings that have to be managed by someone. The good news is that you can leverage GitOps and use Argo CD to manage itself!

All Argo CD settings are persisted in Kubernetes resources. The SSO and RBAC settings stored in ConfigMap and applications and projects are stored in custom resources, so you can store these resources manifests in Git repository and configure Argo CD to use it as a source of truth. This technique is very powerful and allows us to manage configuration settings as well as seamlessly upgrade the Argo CD version.

As a first step let’s demonstrate how to convert SSO and RBAC change we’ve just made imperatively into a declarative configuration. To do so we would need to create a Git repository that stores manifest definitions of every Argo CD component. Instead of starting from scratch, you can just use <https://github.com/gitopsbook/resources> repository that stores code listings and examples of that book. Please navigate to the repository Github URL and create your personal fork so you can store settings specific to your environment.

The required manifest files are located in *chapter-09* directory and the first file we should look at is represented in the listing below.

Listing 9.4 `kustomization.yaml`.

```

apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
  #A

patchesStrategicMerge:
- argocd-cm.yaml          #B
- argocd-rbac-cm.yaml      #C
- argocd-server.yaml       #D

```

#A The remote file URL containing default Argo CD manifests

#B The file path that contains argocd-cm ConfigMap modifications

#C The file path that contains argocd-rbac-cm ConfigMap modifications

#D The file path that contains argocd-server Service modifications

The `kustomization.yaml` file contains references to the default Argo CD manifests and files with the environment specific changes.

The next step is to move your environment specific changes into the Git and push into the remote Git repository. Clone the forked Git repository:

```
$ git clone git@github.com:<USERNAME>/resources.git
```

Repeat changes of `argocd-cm.yaml` and `argocd-rbac-cm.yaml` files described in 9.4.1 and 9.4.2 paragraphs. Add SSO configuration to the ConfigMap manifest in `argocd-cm.yaml`. Update RBAC policy in the `argocd-rbac-cm.yaml` file. Once files are updated commit and push the changes back to the remote repository.

```
$ git commit -am "Update Argo CD configuration"
$ git push
```

The hardest part is done! Argo CD config changes are not version controlled and can be managed using GitOps methodology. The last step is to create an Argo CD application that deploys Kustomize based manifests from your Git repository into the `argocd` namespace:

```

$ argocd app create argocd \
--repo https://github.com/<USERNAME>/resources.git \
--path chapter-09 \
--dest-server https://kubernetes.default.svc \
--dest-namespace argocd \
--sync-policy auto
application 'argocd' created

```

As soon as the application is created the Argo CD should detect already deployed resources and visualize the detected deviations.

So how about managing applications and projects? Both are represented by the Kubernetes custom resource and might be managed using GitOps as well. The manifest below represents the declarative definition of the `sample-app` Argo CD application that we've created manually earlier in the chapter. In order to start managing the `sample-app` declaratively add the `sample-app.yaml` into the resources section of `kustomization.yaml` and push the change back to your repository fork.

Listing 9.5 `sample-app.yaml` .

```

apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: sample-app
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    path: .
  repoURL: https://github.com/<username>/sample-app-deployment

```

As you can see you don't have to choose between declarative and imperative management styles. Argo CD supports using both simultaneously so that some settings are managed using GitOps and some are managed using imperative commands.

9.5 Summary

- Argo CD is designed for enterprise in mind and can be offered as a centralized service to support multi-tenancy and multi-cluster for large enterprises.
- As a continuous deployment tool, Argo CD also provides detail diff among Git, Kubernetes target and running states for observability.
- Argo CD automates three phases in deployment.
 - Retrieve resource manifests
 - Detect and fix the deviations
 - Present the results to end-users.
- Argo CD provides CLI for configuring Application deployment and can be incorporated into CI solutions thru scripting.
- Argo CD's CLI and web interface can be used to inspect applications' sync and health status.
- Argo CD provides resource hooks to enable additional customization of the deployment life cycle.
- Argo CD also provides support to ensure deployment complete and application readiness.
- Argo CD supports both SSO and RBAC integration for enterprise level SSO and access control.

10

Jenkins X

(with **Viktor Farcic & Oscar Medina**)

This chapter covers:

- What is Jenkins X
- Installation of Jenkins X
- Importing projects into Jenkins X
- Promoting a release to production in Jenkins X

In this chapter you will learn how to use the Jenkins X to deploy our reference example application to Kubernetes. You will also learn how Prow, Jenkins X Pipeline Operator and Tekton work together to build a CI/CD pipeline.

We recommend you read chapters 1, 2, 3, and 5 before reading this chapter.

10.1 What Is Jenkins X?

To understand the intricacies and inner workings of Jenkins X, we need to understand Kubernetes. However, you do not need to understand Kubernetes to use Jenkins X. That is one of the main contributions of the project. Jenkins X allows us to harness the power of Kubernetes without spending an eternity learning the ever-growing list of the things Kubernetes does. Jenkins X⁷⁵ is an open source-tool that simplifies the complex processes into concepts that can be adopted quickly and without spending months trying to figure out “the right way to do stuff.” It helps by removing and simplifying some of the problems caused by the overall complexity of Kubernetes and its ecosystem. If you are indeed a Kubernetes ninja, you will appreciate all the effort put into Jenkins X. If you’re not, you will be able to jump right in and harness the power of Kubernetes without ripping your hair out

⁷⁵ <https://jenkins-x.io/>

of frustration caused by Kubernetes complexity. In section 10.2, we will discuss the patterns and tools and in detail.

NOTE Jenkins X is a free open source tool with enterprise support offered by Cloudbees

Today, most software vendors are building their next generation of software to be Kubernetes-native or, at least, to work better inside it. A whole ecosystem is emerging and treating Kubernetes as a blank canvas. As a result, new tools are being added on a daily basis, and it is becoming evident that Kubernetes offers near-limitless possibilities. However, with that comes increased complexity. It is harder than ever to choose which tools to use. How are we going to develop our applications? How are we going to manage different environments? How are we going to package our applications? Which process are we going to apply for application life cycles? And so on and so forth. Assembling a Kubernetes cluster with all the tools and processes takes time, and learning how to use what we assembled feels like a never-ending story. Jenkins X aims to remove those and other obstacles.

Jenkins X is opinionated. It defines many aspects of the software development lifecycle, and it makes decisions for us. It tells us what to do and how. It is like a tour guide on your vacation that shows you where to go, what to look at, when to take a photo, and when it's time to take a break. At the same time, it is flexible and allows power users to tweak it to fit their own needs.

The real power behind Jenkins X is the process, the selection of tools, and the glue that wraps everything into one cohesive unit that is easy to learn and use. We (people working in the software industry) tend to reinvent the wheel all the time. We spend countless hours trying to figure out how to develop our applications faster and how to have a local environment that is as close to production as possible. We dedicate time searching for tools that will allow us to package and deploy our applications more efficiently. We design the steps that form a continuous delivery pipeline. We write scripts that automate repetitive tasks. And yet, we cannot escape the feeling that we are likely reinventing things that were already done by others. Jenkins X is designed to help us with those decisions, and it helps us to pick the right tools for a job. It is a collection of the industry's best practices. In some cases, Jenkins X is the one defining those practices, while in others, it helps us in adopting those made by others.

If we are about to start working on a new project, Jenkins X will create the structure and the required files. If we need a Kubernetes cluster with all the tools selected, installed, and configured, Jenkins X will do that. If we need to create Git repositories, set webhooks⁷⁶, and create continuous delivery pipelines, all we need to do is execute a single `jx` command. The list of what Jenkins X does is vast, and it grows every day.

NOTE Jenkins vs. Jenkins X If you are familiar with Jenkins, you need to clear your mind from any Jenkins experience you might already have. Sure, Jenkins is there, but it is only a part of the package. Jenkins X is very different from the "traditional Jenkins". The differences are so massive that the only way for you to embrace it is to forget what you know about Jenkins and start from scratch.

⁷⁶ <https://developer.github.com/webhooks/>

10.2 Exploring Prow, Jenkins X Pipeline Operator, And Tekton

The serverless flavor of Jenkins X or, as some call it, Jenkins X Next Generation, is an attempt to redefine how we do continuous delivery and GitOps inside Kubernetes clusters. It does that by combining quite a few tools into a single easy-to-use bundle. As a result, most people will not have a need to understand intricacies of how the pieces work independently, nor how they are all integrated. Instead, many will merely push a change to Git and let the system do the rest. But, there are always those who would like to know what's happening behind the hood. To satisfy those craving for insight, we'll explore the processes and the components involved in the serverless Jenkins X platform. Understanding the flow of an event initiated by a Git webhook will give us insight into how the solution works and help us later on when we go deeper into each of the new components.

Everything starts with a push to a Git repository, which in turn, sends a webhook request to the cluster. Where things differ from traditional Jenkins setup is that there is no Jenkins to accept those requests. Instead, we have Prow⁷⁷. It does quite a few things, but, in the context of webhooks, its job is to receive requests and decide what to do next. Those requests are not limited only to push events, but also include slash commands (such as /approve) we can specify through pull request comments.

⁷⁷ <https://github.com/kubernetes/test-infra/tree/master/prow>

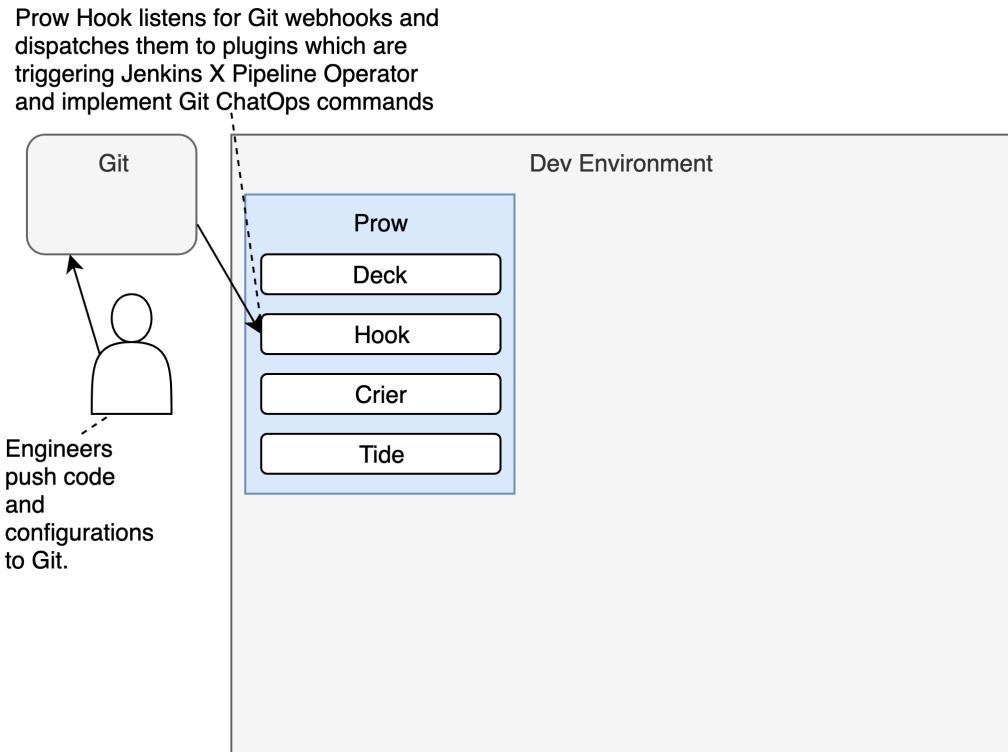


Figure 10.1 Engineers push code and configurations to Git. Prow Hook listens for Git we hooks and dispatches them to plugins.

Prow consists of a few distinct components (Deck, Hook, Crier, Tide, and more). However, we won't go into the roles of each of them. For now, the vital thing to note is that Prow is the entry point to the cluster. It receives Git requests generated either by Git actions (e.g., push) or through slash commands in comments.

Prow might do quite a few things upon receiving a request. If it comes from a command from a Git comment, it might re-run tests, merge a pull request, assign a person, or one of the many other Git related actions. If a webhook informs it that a new push was made, it will send a request to the Jenkins X Pipeline Operator that will make sure that a build corresponding to a defined pipeline is run. Finally, Prow also reports the status of a build back to Git.

Those features are not the only types of actions Prow might perform but, for now, you probably got the general gist. Prow is in charge of communication between Git and the processes inside our cluster.

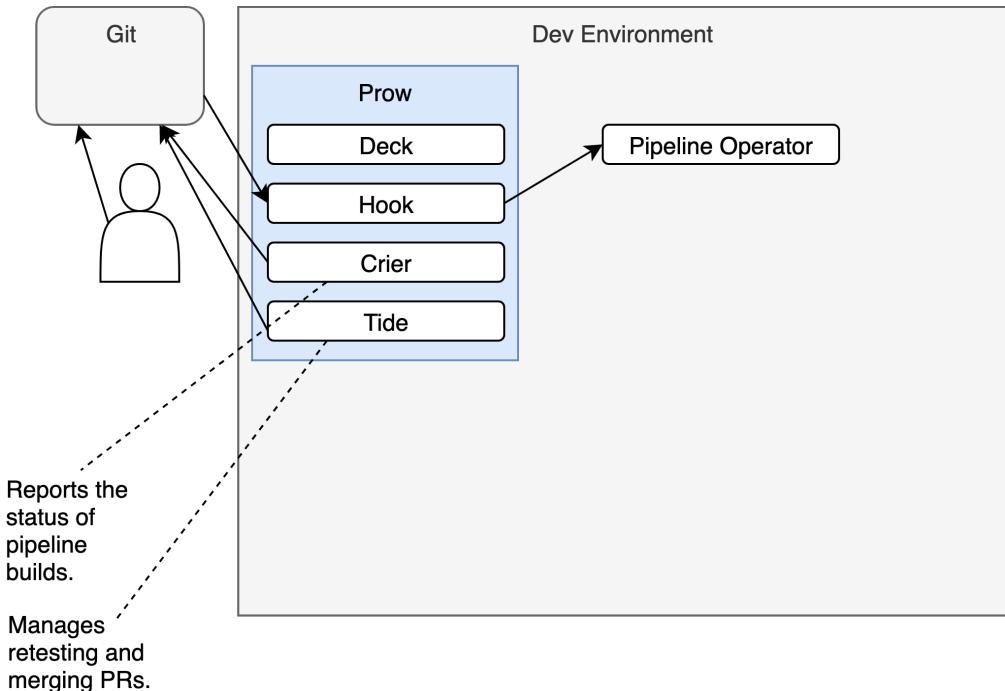


Figure 10.2 When a Prow hook receives a request from a Git we hook, it forwards it to Jenkins X Pipeline Operator.

The role of the operator is to fetch the `jenkins-x.yml` file from the repository that initiated the process and to transform it into Tekton Tasks and Pipelines. They, in turn, define the complete pipeline that should be executed as a result of pushing a change to Git.

NOTE **Tekton**⁷⁸Tekton is a Kubernetes-native open-source framework for creating continuous integration and delivery (CI/CD) systems

⁷⁸<https://cloud.google.com/tekton>

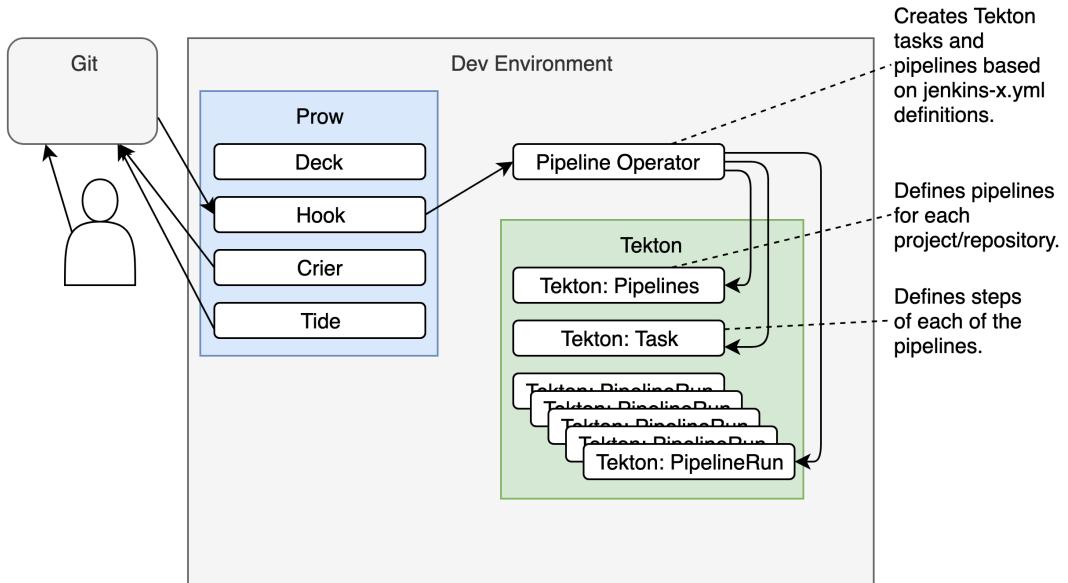


Figure 10.3 The Pipeline Operator is to simplify definitions of our continuous delivery processes, and Tekton does the heavy lifting to define pipelines for each project/repository,

Tekton is a very low-level solution and is not meant to be used directly. Writing Tekton definitions can be quite painful and complicated. Pipeline Operator simplifies that through easy to learn and use YAML format for defining pipelines. Following is an example of what the base pipeline will provide

NOTE As you will discover in Section 10.3, “Importing Projects into Jenkins X”, the pipeline file for your project will be called `jenkins-x.yml`, which includes a single line “`buildPack: go`” to reference the following pipeline file. If you like to learn more about how pipelines work, please refer to the Jenkins X documentation⁷⁹.

Listing 10.1 `pipeline.yaml` .

```

extends:
  import: classic
  file: go/pipeline.yaml
pipelines:
  pullRequest:
    build:
      steps:
        - sh: export VERSION=$PREVIEW_VERSION && skaffold build -f skaffold.yaml
          name: container-build
    postBuild:
      steps:

```

⁷⁹ <https://jenkins-x.io/docs/reference/components/build-packs/>

```

- sh: jx step post build --image $DOCKER_REGISTRY/$ORG/$APP_NAME:$PREVIEW_VERSION
  name: post-build
promote:
  steps:
  - dir:
    /home/jenkins/go/src/REPLACE_ME_GIT_PROVIDER/REPLACE_ME_ORG/REPLACE_ME_APP_NAME/char
    ts/preview
    steps:
    - sh: make preview
      name: make-preview
    - sh: jx preview --app $APP_NAME --dir ../../..
      name: jx-preview

release:
build:
  steps:
  - sh: export VERSION=`cat VERSION` && skaffold build -f skaffold.yaml
    name: container-build
  - sh: jx step post build --image $DOCKER_REGISTRY/$ORG/$APP_NAME:\$(cat VERSION)
    name: post-build
promote:
  steps:
  - dir:
    /home/jenkins/go/src/REPLACE_ME_GIT_PROVIDER/REPLACE_ME_ORG/REPLACE_ME_APP_NAME/char
    ts/REPLACE_ME_APP_NAME
    steps:
    - sh: jx step changelog --version v\$(cat ..../VERSION)
      name: changelog
    - comment: release the helm chart
      name: helm-release
      sh: jx step helm release
    - comment: promote through all 'Auto' promotion Environments
      sh: jx promote -b --all-auto --timeout 1h --version \$(cat ..../VERSION)
      name: jx-promote

```

Tekton creates a PipelineRun for each build initiated by each push to one of the associated branches (ex: Master branch, PRs). It performs all the steps we need to validate a push. It runs tests, stores binaries in registries (ex: Docker Registry, Nexus, and ChartMuseum), and it deploys a release to a temporary (PR) or a permanent (staging or production) environment.

The complete flow can be seen in the diagram that follows.

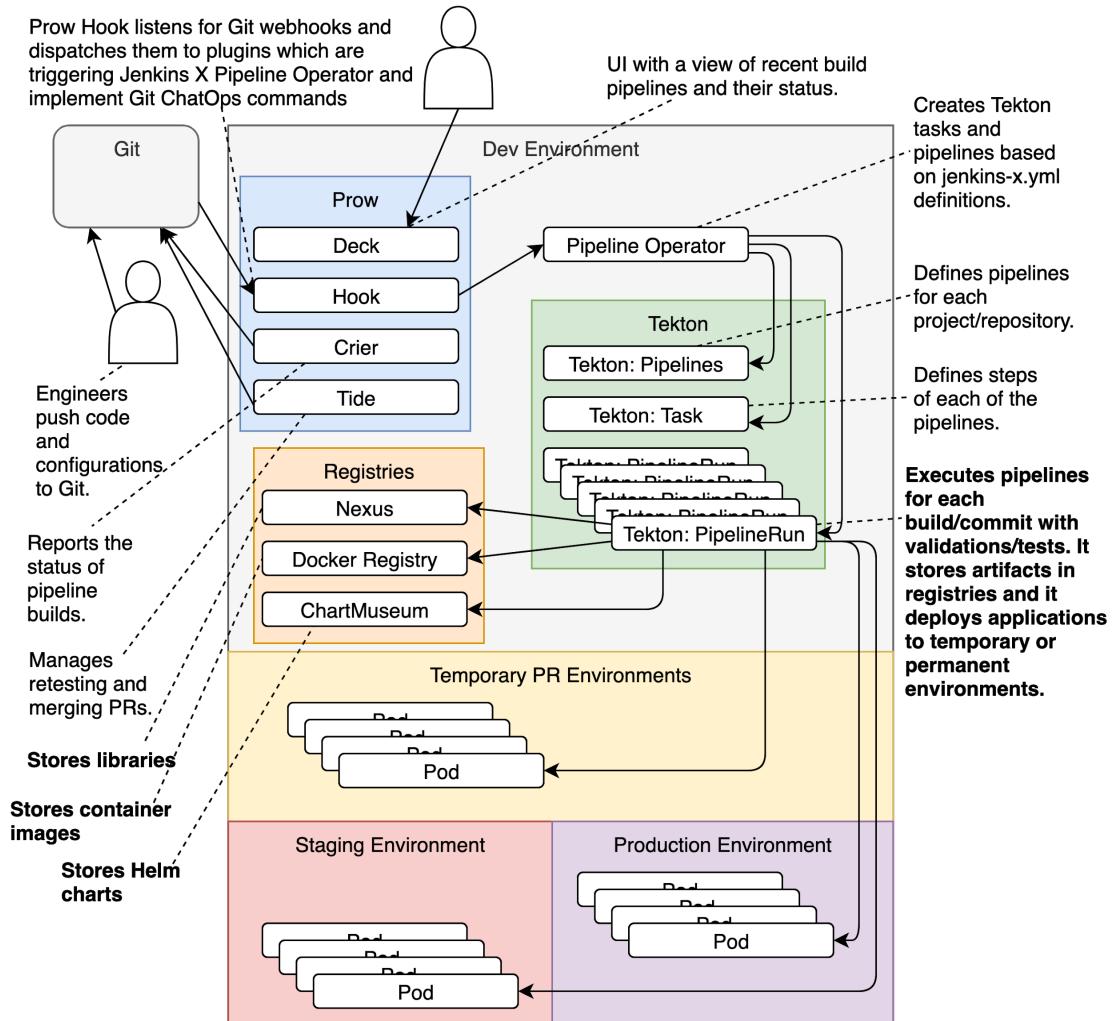


Figure 10.4 The complete flow of events starts from PR, webhook in Prow, Pipeline Operator to Tekton. Jenkins X will execute pipelines for each build/commit and deploy applications.

Exercise 10.1

Which component will receive the Git Web Hook request?

Which component will orchestrate the deployment?

10.3 Importing Projects Into Jenkins X

You can see how we can fast-track the development and continuous delivery of new applications with Jenkins X quickstarts. However, it is likely that your company was not

formed yesterday. That means that you already have some apps, and hopefully, you'd like to move them to Jenkins X.

From a Jenkins X perspective, importing an existing project is relatively straightforward. All we have to do is execute `jx import`, and Jenkins X will do its magic. It will create the files we need. If we do not yet have `skaffold.yml`, it will be generated for us. If we did not create a Helm chart, it would create that as well. No `Dockerfile`? No problem. We'll get that as well. Never wrote a Jenkins pipeline for that project? Again, that is not an issue. We'll get a `jenkins-x.yml` file that is automatically generated. Jenkins X will reuse the things we already have, and create those that we're missing.

The import process does not limit itself to creating missing files and pushing them to Git. It'll also create a job in Jenkins, webhooks in GitHub, and quite a few other things.

NOTE Please refer to Appendix B for more information on how to install Jenkins X.

10.3.1 Importing A Project

We'll import the application stored in the `gitopsbook/sample-app` repository. We'll use it as a guinea pig for testing the import process as well as to flesh out potential problems we might encounter.

But, before we import the repository, you'll have to fork the code. Otherwise, you won't be able to push changes since you are not (yet) a collaborator on that specific repository.

```
$ open "https://github.com/gitopsbook/sample-app"
```

Please make sure that you are logged in and click the *Fork* button located in the top-right corner. Follow the on-screen instructions.

Next, we need to clone the repository you just forked.

```
$ GH_USER=[...] #A
$ git clone https://github.com/$GH_USER/sample-app.git
$ cd sample-app
```

#A Please replace ` [...]` with your GitHub user before executing the commands that follow.

Now you should have the intended code in the master branch of the repository you forked. Feel free to take a look at what we have by opening the repository in a browser. Fortunately, there is a `jx` command that does just that.

```
$ jx repo --batch-mode
```

Let's quickly explore the files of the project, before we import it into Jenkins X.

```
$ ls -1
Dockerfile
Makefile
README.md
main.go
```

As you can see, there's (almost) nothing in that repository but Go⁸⁰ code (*.go).

That project is one extreme of the possible spectrum of projects we might want to import to Jenkins X. It only has the code of the application. There is a `Dockerfile`. However, there is no Helm chart or even a script for building a binary, nor is there a mechanism to run tests, and there is definitely no `jenkins-x.yml` file that defines a continuous delivery pipeline for the application. There's only code, and (almost) nothing else.

Such a situation might not be your case. Maybe you do have scripts for running tests or building the code. Or perhaps you are already a heavy Kubernetes user, and you do have a Helm chart. You might have other files as well. We'll discuss those situations later. For now, we'll work on the case when there is nothing but the code of an application.

Let's see what happens when we try to import that repository into Jenkins X.

```
$ jx import
intuitdep954b9:sample-app byuen$ jx import
WARNING: No username defined for the current Git server!
? github.com username: billyy #A
To be able to create a repository on github.com we need an API Token
Please click this URL and generate a token #B
https://github.com/settings/tokens/new?scopes=repo,read:user,read:org,user:email,write:repo
_hook,delete_repo
```

Then COPY the token and enter it below:

```
? API Token: *****
performing pack detection in folder /Users/byuen/git/sample-app
--> Draft detected Go (48.306595%)
selected pack: /Users/byuen/.jx/draft/packs/github.com/jenkins-x-buildpacks/jenkins-x-
    kubernetes/packs/go
replacing placeholders in directory /Users/byuen/git/sample-app
app name: sample-app, git server: github.com, org: billyy, Docker registry org: hazel-
    charter-283301
skipping directory "/Users/byuen/git/sample-app/.git"
Draft pack go added
? Would you like to define a different preview namespace? No #C
Pushed Git repository to https://github.com/billyy/sample-app.git
Creating GitHub webhook for billyy/sample-app for url http://hook-
    jx.34.74.32.142.nip.io/hook
Created Pull Request: https://github.com/billyy/environment-cluster-1-dev/pull/1
Added label updatebot to Pull Request https://github.com/billyy/environment-cluster-1-
    dev/pull/1
created pull request https://github.com/billyy/environment-cluster-1-dev/pull/1 on the
    development git repository https://github.com/billyy/environment-cluster-1-dev.git
regenerated Prow configuration
PipelineActivity for billyy-sample-app-master-1
upserted PipelineResource meta-billyy-sample-app-master-cdxm7 for the git repository
    https://github.com/billyy/sample-app.git
upserted Task meta-billyy-sample-app-master-cdxm7-meta-pipeline-1
upserted Pipeline meta-billyy-sample-app-master-cdxm7-1
created PipelineRun meta-billyy-sample-app-master-cdxm7-1
created PipelineStructure meta-billyy-sample-app-master-cdxm7-1

Watch pipeline activity via: jx get activity -f sample-app -w
Browse the pipeline log via: jx get build logs billyy/sample-app/master
```

⁸⁰ <https://golang.org/>

```
You can list the pipelines via: jx get pipelines
When the pipeline is complete: jx get applications
```

For more help on available commands see: <https://jenkins-x.io/developing/browsing/>

Note that your first pipeline may take a few minutes to start while the necessary images get downloaded!

```
#A Github username
#B Generate new token
#C Default "No" for preview namespace
```

We can see from the output that Jenkins X detected that the project is 100% written in Go, so it selected the go build pack. It applied it to the local repository and pushed the changes to GitHub. Furthermore, it created a Jenkins project as well as a GitHub webhook that will trigger builds whenever we push changes to one of the selected branches. Those branches are by default master, develop, PR-.*, and feature.*. We could have changed the pattern by adding the --branches flag. But, for our purposes, and many others, those branches are just what we need.

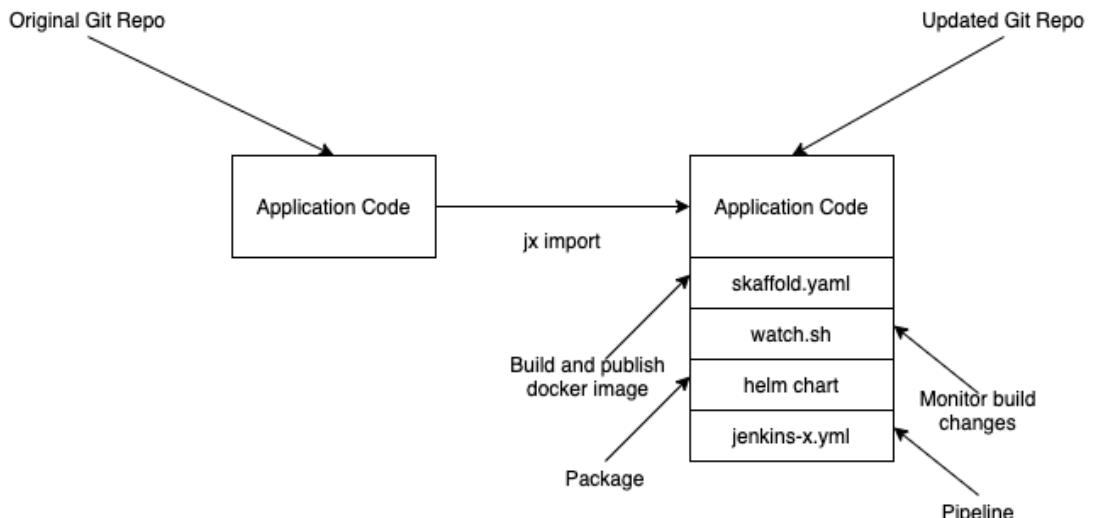


Figure 10.5 Files added by `jx import`

Now, let's take another look at the files in the local copy of the repository.

```
$ ls -1
Dockerfile
Makefile
OWNERS
OWNERS_ALIASES
README.md
charts
jenkins-x.yml
```

```
main.go
skaffold.yaml
watch.sh
```

We can see that quite a few new files were added to the project through the import process. We have a `Dockerfile` that will be used to build container images, and we have a `jenkins-x.yml` that defines all the steps of our pipeline.

We also got a `Makefile` that defines targets to build, test, and install the application. There is also the `charts` directory that contains files in Helm format for packaging, installing, and upgrading our application. We also got `watch.sh` which monitors build changes and invokes `skaffold.yaml`. `skaffold.yaml` contains the instruction to build and publish the container images. There are a few other new files (ex: `OWNERS`) added to the mix.

Now that the project is in Jenkins X, we should see it as one of the activities and observe the first build in action. You already know that we can limit the retrieval of Jenkins X activities to a specific project and that we can use `--watch` to watch the progress.

NOTE Please wait until the 'jx promote' and 'PullRequest' complete before proceeding to the rest of the tutorial. If the process takes more than 60 minutes, the Pull Request will show "Failed" status. If you check [github](#), the PR will still merge after 'jx promote' completes.

```
$ jx get activities --filter sample-app --watch
STEP                                              STARTED AGO DURATION STATUS
billyy/sample-app/master #1
  Version: 0.0.1
  meta pipeline
    Credential Initializer
    Working Dir Initializer
    Place Tools
    Git Source Meta Billyy Sample App Master R Xnf14 Vrvtm
      https://github.com/billyy/sample-app.git
    Setup Builder Home
    Git Merge
    Merge Pull Refs
    Create Effective Pipeline
    Create Tekton Crds
  from build pack
    Credential Initializer
    Working Dir Initializer
    Place Tools
    Git Source Billyy Sample App Master Releas 658x6 Nzdbp
      https://github.com/billyy/sample-app.git
    Setup Builder Home
    Git Merge
    Setup Jx Git Credentials
    Build Make Build
    Build Container Build
    Build Post Build
    Promote Changelog
    Promote Helm Release
    Promote Jx Promote
  Promote: staging
    PullRequest
      PullRequest: https://github.com/billyy/environment-cluster-1-staging/pull/1
```

chore: sample-app to 0.0.1 #1

Merged billyy merged 1 commit into `master` from `promote-sample-app-0.0.1` 11 hours ago

Conversation 1 Commits 1 Checks 0 Files changed 2

billyy commented 12 hours ago

chore: Promote sample-app to version 0.0.1

chore: Promote sample-app to version 0.0.1 ✓ 7d73b8f

billyy added updatebot size/S Owner labels 12 hours ago

billyy commented 12 hours ago

[APPROVALNOTIFIER] This PR is NOT APPROVED

This pull-request has been approved by:
To complete the [pull request process](#), please assign
You can assign the PR to them by writing `/assign` in a comment when ready.

The full list of commands accepted by this bot can be found [here](#).

▼ Details
Needs approval from an approver in each of these files:
• OWNERS

Approvers can indicate their approval by writing `/approve` in a comment
Approvers can cancel approval by writing `/approve cancel` in a comment

Figure 10.6 Jenkins X will generate a Pull Request to add a new version of our app. This is GitOps at work!

The pipeline activities give you a lot of detail on the pipeline stages and steps. However, one of the most important details is the PR that gets merged into the staging environment. This tells Jenkins X to add that new version of our app into the `env/requirements.yaml` file. This is GitOps at work!

So far, Jenkins X created the files it needs, it created a GitHub webhook, it created a pipeline, and it pushed changes to GitHub. As a result, we got our first build, and by the look of it, it was successful. But let's double-check that everything is OK.

Please open the PullRequest link on your browser by clicking on the link from the activity output, as shown in Figure 10.5.

So far, so good. The `sample-app` job created a pull request to the `environment-cluster-1-staging` repository. As a result, the webhook from that repository should have initiated a

pipeline activity, and the result should be a new release of the application in the staging environment. We won't go through that part of the process just yet. For now, just note that the application should be running, and we'll check that soon.

The information we need to confirm that the application is indeed running is in the list of the applications running in the staging environment. We'll explore the environments later. For now, just run the command that follows.

```
$ jx get applications
APPLICATION STAGING PODS URL
sample-app 0.0.1      http://sample-app-jx-staging.34.74.32.142.nip.io
```

We can see the address through which our application should be accessible in the URL column. Please copy it and use it instead of [...] in the command that follows.

```
$ STAGING_ADDR=[...]
$ curl "$STAGING_ADDR/demo/hello"          #A
Kubernetes ♡ Golang!
```

#A Staging URL address

The output shows Kubernetes ♡ Golang!, thus confirming that the application is up-and-running and that we can reach it.

Before we proceed, we'll go out of the sample-app directory.

We reached the final stage, at least from the application lifecycle point of view.

NOTE For a complete application lifecycle reference, please refer to figure 4.6 in chapter 4 (pipelines).

Actually, we skipped creating a pull request, which happens to be one of the most important features of Jenkins X. Nevertheless, we do not have enough space to cover all Jenkins X features, so I'll leave PRs and others for you to discover on your own (PR can be found from output of the `jx get activities` command above). For now, we'll focus on the final stage of the application lifecycle by exploring promotions to production. We've already covered the following:

1. We saw how to import an existing project and how to create a new one.
2. We saw how to develop build packs that will simplify those processes for the types of applications that are not covered with the existing build packs or for those that deviate from them.
3. Once we added our app to Jenkins X, we explored how it implements GitOps processes through environments (e.g., staging and production).
4. Then we moved into the application development phase and explored how DevPods help us to set a personal application-specific environment that simplifies the "traditional" setup that forced us to spend countless hours setting it on our laptop and, at the same time, that avoids the pitfalls of shared development environments.
5. Once the development of a feature, a change, or a bug fix is finished, we created a pull request, we executed automatic validations, and we deployed the release candidate to a PR-specific preview environment so that we can check it manually as well. Once we were satisfied with the changes we made, we merged it to the master

branch, and that resulted in deployment to the environments set to receive automatic promotions (e.g., staging) as well as in another round of testing. Now that we are comfortable with the changes we did, all that's left is to promote our release to production.

The critical thing to note is that promotion to production is not a technical decision. By the time we reach this last step in the software development lifecycle, we should already know that the release is working as expected. We already gathered all the information we need to make a decision to go live. Therefore, the choice is business-related. "When do we want our users to see the new release?" We know that every release that passes all the steps of the pipeline is production-ready, but we do not know when to release it to our users. But, before we discuss when to release something to production, we should decide who does that. The actor will determine when is the right time. Does a person approve a pull request, or is it a machine?

Business, marketing, and management might be decision-makers in charge of promotion to production. In that case, we cannot initiate the process when the code is merged to the master branch (as with the staging environment), and that means that we need a mechanism to start the process manually through a command. If executing a command is too complicated and confusing, it should be trivial to add a button (we'll explore that through the UI later). There can also be the case when no one makes a decision to promote something to production. Instead, we can promote each change to the master branch automatically. In both cases, the command that initiates the promotion is the same. The only difference is in the actor that executes it. Is it us (humans) or Jenkins X (machines)?

At the moment, our production environment is set to receive manual promotions. As such, we are employing continuous delivery that has the whole pipeline fully automated and requires a single manual action to promote a release to production. All that's left is to click a button or, as is our case, to execute a single command. We could have added the step to promote production to `Jenkinsfile`, and in that case, we'd be practicing continuous deployment (not delivery). That would result in a deployment of every merge or push to the master branch. But, we aren't practicing continuous deployment today, and we'll stick with the current setup and jump into the last stage of continuous delivery. We'll promote our latest release to production.

10.3.2 Promoting A Release To The Production Environment

Now that we feel that our new release is production-ready, we can promote it to production. But, before we do that, we'll check whether we already have something running in production.

```
$ jx get applications --env production
APPLICATION
sample-app
```

How about staging? We must have the release of our `sample-app` application running there. Let's double-check.

```
$ jx get applications --env staging
APPLICATION STAGING PODS URL
```

```
sample-app 0.0.1 1/1 http://sample-app-jx-staging.34.74.32.142.nip.io
```

For what we're trying to do, the important piece of the information is the version displayed in the STAGING column.

Now we can promote the specific version of *sample-app* to the production environment.

```
$ VERSION=[...]
$ jx promote sample-app --version $VERSION --env production --batch-mode
```

#A Before executing the command that follows, please make sure to replace [...] with the version from the STAGING column from the output of the previous command.

It'll take a minute or two until the promotion process is finished. You can use the following command again to monitor the status.

```
$ jx get activities
...
Promote: production
  PullRequest
    PullRequest: https://github.com/billyy/environment-cluster-1-production/pull/2 Merge
    SHA: 33b48c58b3332d3abc2b0c4dcaba8d7ddc33c4b3
  Update
  Promoted
Application is at: http://sample-app-jx-production.34.74.32.142.nip.io
```

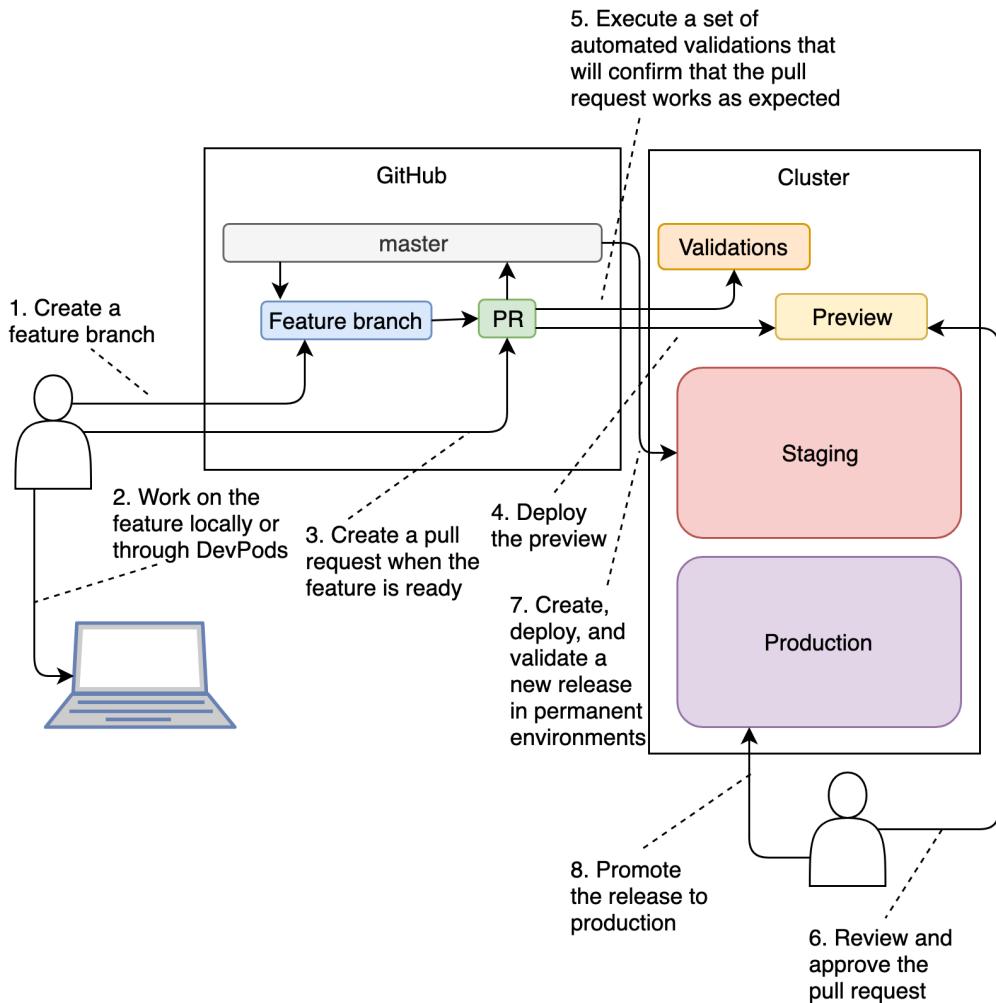


Figure 10.7 The `jx promote` command will create a new branch in the production environment as well as deploy to preview. At the end of the command execution, the new release will be promoted to production.

The command we just executed will create a new branch in the production environment (`environment-pisco-sour-production`). Further on, it'll follow the same practice based on pull requests as the one employed in anything else we did so far. It'll create a pull request and wait until a Jenkins X build is finished and successful. You might see errors stating that it failed to query the Pull Request. That's normal. The process is asynchronous, and `jx` is periodically querying the system until it receives the information that confirms that the pull request was processed successfully.

Once the pull request is processed, it'll be merged to the master branch, and that will initiate yet another Jenkins X build. It'll run all the steps we defined in the repository's `Jenkinsfile`. By default, those steps are only deploying the release to production, but we could have added additional validations in the form of integration or other types of tests. Once the build initiated by the merge to the master branch is finished, we'll have the release running in production, and the final output will state that merge status checks all passed, so the promotion worked!

The process of manual promotion (ex: production) is the same as the one we experienced through automated promotions (ex: staging). The only difference is who executes promotions. Those that are automated are initiated by application pipelines pushing changes to Git. On the other hand, manual promotions are triggered by us (humans).

Next, we'll confirm that the release is indeed deployed to production by retrieving all the applications in that environment.

```
$ jx get applications --env production
APPLICATION PRODUCTION PODS URL
sample-app 0.0.1 http://sample-app-jx-production.35.185.219.24.nip.io
```

In our case, the output states that there is only one application (`sample-app`) running in production and that the version is 0.0.1.

To be on the safe side, we'll send a request to the release of our application running in production.

```
$ PROD_ADDR=[...]
$ curl "$PROD_ADDR/demo/hello" #A
Kubernetes ♥ Golang!
```

#A Before executing the commands that follow, please make sure to replace [...] with the URL column from the output of the previous command.

10.4 Summary

- Jenkins X defines the process, the selection of tools, and the glue that wraps everything into one cohesive unit that is easy to learn and use.
- Prow provides GitHub automation in the form of policy enforcement and automatic PR merging.
- Pipeline Operator is to orchestrate and simplify definitions of our continuous delivery processes.
- Tekton is a Kubernetes-native open-source framework for creating continuous integration and delivery (CI/CD) systems.
- To import projects into Jenkins X, you just need to execute 'jx import' which will add all the necessary files to your repo and create the pipelines and environments.
- To promote release into production environments, you can simply execute the 'jx promote' command which will generate PR to add the new release, deploy to preview for testing and promote (deploy) to production.

11

Flux

This chapter covers:

- **What is Flux?**
- **Deploying an application using Flux**
- **Setting up Multi-tenancy with Flux**

In this chapter you will learn how to use the Flux GitOps operator to deploy our reference example application to Kubernetes. You will also learn how Flux can be used as part of a multi-tenancy solution.

We recommend you read chapters 1, 2, 3, and 5 before reading this chapter.

11.1 What is Flux?

Flux is an open-source project that implements GitOps driven continuous deployment for Kubernetes. The project was started in 2016 at Weaveworks⁸¹ and joined CNCF Sandbox three years later.

NOTE **CNCF**Linux Foundation project that hosts critical components of the global technology infrastructure.

Notable, Weaveworks is that same company that coined the term GitOps term itself. Along with other great open source projects for Kubernetes, the company formulated GitOps best practices and contributed a lot to GitOps evangelizing. The Flux evolution illustrates how the idea of GitOps was evolved over time based on practical experience into its current form.

The Flux project was created to automate container image delivery to Kubernetes and fill the gap between the Continuous Integration and Continuous Deployment processes. The workflow described in the project introduction blog is focused on Docker registry scanning,

⁸¹ <https://www.weave.works/>

calculating the latest image version, and promoting it to the production cluster. After several iterations, the Flux team has realized all the benefits of a Git-centric approach. Before publishing the v1.0 release, the project architecture was reworked to use Git as the source of truth and formulated the main phases of the GitOps workflow.

11.1.1 What Flux Does?

Flux is laser-focused on automated manifest delivery to the Kubernetes cluster. The project is probably the least opinionated GitOps operation among other operators described in this book. Flux does not introduce any additional layers on top of Kubernetes, such as application or own access control system. A single Flux manages one Kubernetes cluster and requires the user to maintain one Git repository that represents the cluster state. Instead of introducing user management, SSO integration, and own access control, Flux typically runs inside of the managed cluster and relies on Kubernetes RBAC. This approach significantly simplifies the Flux configuration and helps flatten the learning curve.

NOTE RBAC Kubernetes supports role-based access control (RBAC), which allows containers to be bound to roles which give them permissions to operate on various resources

The simplicity of Flux also makes it virtually maintenance-free and simple to integrate into cluster bootstrapping since no new component or admin privilege is required. With the Flux command-line interface, Flux deployment can be easily incorporated into the cluster provisioning scripts to enable automated cluster creation.

Flux is not limited only to cluster bootstrapping. It is successfully used as a continuous deployment tool for applications. In the multi-tenant environment, each team can install an instance of Flux with limited access and use it to manage a single namespace. That fully empowers the team to manage resources in application namespace and is still one hundred percent secure because Flux access is managed by Kubernetes RBAC.

The simplicity of the project brings advantages and disadvantages that are viewed differently by the different teams. One of the most important considerations is that Flux has to be configured and maintained by the Kubernetes end user. That implies that the team gets more power but also has more responsibility. The alternate approach, which is taken by Argo CD, is to provide GitOps function as a service.

11.1.2 Docker Registry Scanning

In addition to the core functionality of GitOps, the project offers one more notable feature. Flux is able to scan the Docker registry and automatically update images in the deployment repository when new tags get pushed into the registry. Although this functionality is not a core GitOps feature, it simplifies the life of developers and increases productivity. Let's consider the developer workflow without automated deployment repository updates.

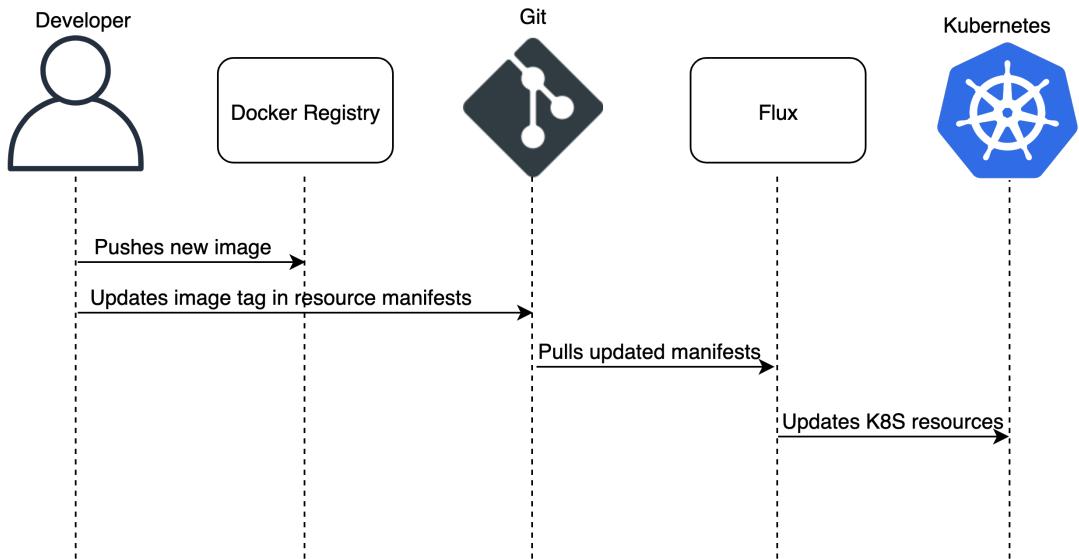


Figure 11.1 Developer pushes new images manually using continuous integration tools, and then updates the deployment Git repository with a new image's tag. Flux notices the manifest change in Git and propagates it to the Kubernetes cluster.

The developer teams often complain about the second step, because it requires manual work, and try to automate it. Typically the solution is to automate manifest updates using the CI pipeline. The CI approach solves the problem but requires scripting and might be fragile.

Flux goes one step ahead and automates the deployment repository updates. Instead of using the CI system and scripting, you can configure Flux to automatically update the deployment repository every time when a new image is pushed to the Docker registry. The developer's workflow with automated Docker registry scanning is represented in the following picture.

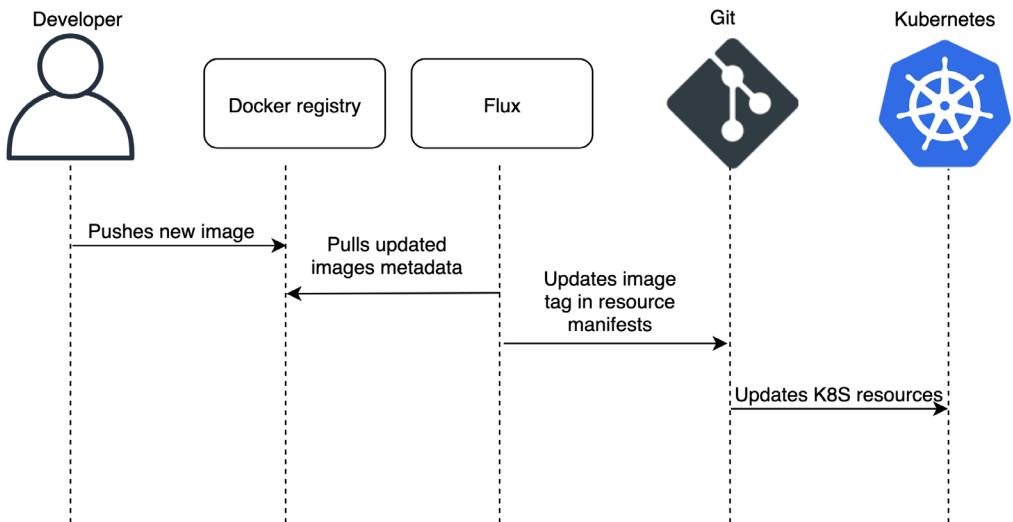


Figure 11.2 When the automated repository updates are enabled, Flux takes full control over both deployment repository and Kubernetes cluster management.

The only developer's responsibility is to make a code change and let the CI system push the updated Docker image into the registry. The automated deployment repository management is especially useful if the image tags are following semantic versioning convention.

NOTE Semantic Versioning⁸² Semantic Versioning is a formal convention for specifying compatibility using a three-part version number: major version, minor version, and patch.

Flux allows configuring the image tag filter that leverages the semantic version convention. The typical use case is to automate minor and patch releases that are supposed to be safe and backward compatible and manually deploy major releases.

The obvious benefit of the Docker registry scanning feature, compared to using the Continuous Integration pipeline, is that you don't have to spend time to implement the repository update step in your pipeline. The convenience, however, comes with more responsibility. Incorporating a deployment repository update into the Continuous Integration pipeline gives full control and allows us to run more tests after an image is pushed to the Docker registry. If the Flux Docker registry scanning is enabled, you have to make sure that the image is very well tested before pushing it to the Docker registry to avoid accidentally deploying to production.

Exercise 11.1

Consider the advantages and disadvantages of the Docker registry monitoring feature and try to answer if it is suitable for your team.

⁸² <https://semver.org/>

11.1.3 Architecture

Flux consists of only two components: Flux daemon and key-value store Memcached⁸³.

NOTE Memcached Memcached is an open-source high-performance, distributed memory object caching system.

The Flux architecture diagram is represented in the figure below:

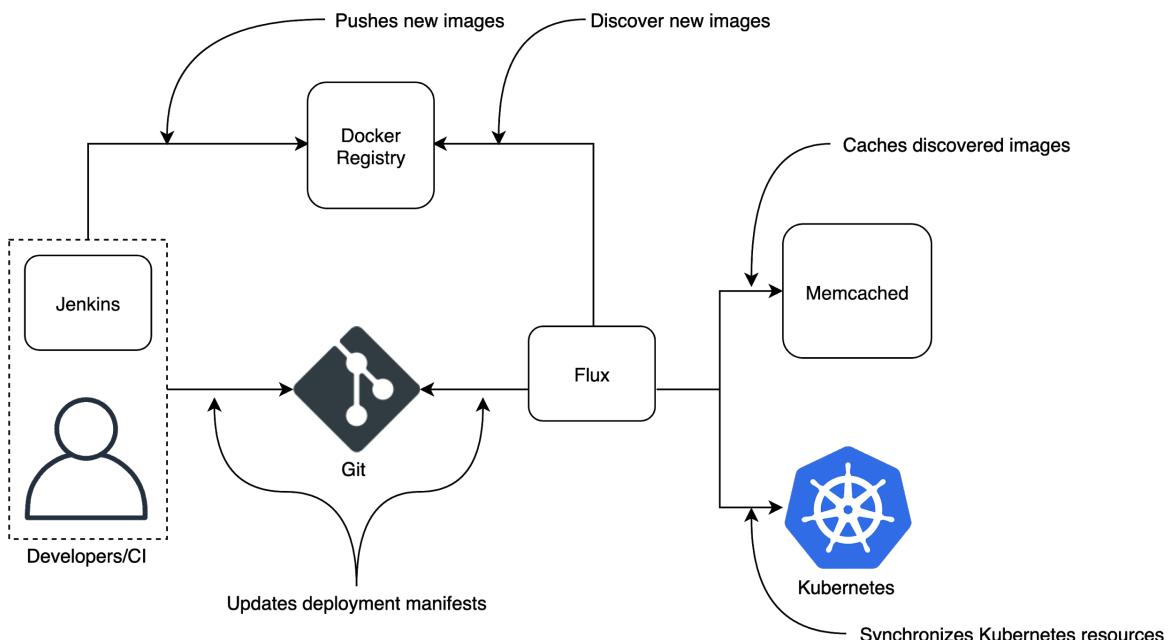


Figure 11.3 The Flux daemon is the main component that is responsible for the majority of Flux features. It clones the Git repository, generates manifests, propagates changes to the Kubernetes cluster, and scans to Docker registry.

There must be only one replica of the Flux daemon running at the same time. This is not an issue, however, because even if daemon crashes in the middle of a deployment, it restarts quickly and idempotently resumes the deployment process.

The main purpose of Memcached is to support Docker registry scanning. Flux uses it to store a list of available image versions of each Docker image. The Memcached deployment is an optional component and not required unless you want to use the Docker registry scanning feature. To remove it just use `--registry-disable-scanning` flag during the installation step.

⁸³ <https://memcached.org/>

Exercise 11.2

Which component logs should you check to troubleshoot deployment issues?

11.2 Simple application deployment

We've learned a lot about Flux already, and now it is time to see it in action. First of all, we need to get it running. The Flux installation consists of two steps: installing the Flux CLI and configuring the demon in your cluster. Use Appendix B to learn how to install `fluxctl` and get ready to deploy your first application.

11.2.1 Deploying the first application

The `fluxctl` and `minikube` applications are only two components required to start managing Kubernetes resources with Flux. The next step is to prepare the Git repository with the Kubernetes manifests. The manifests for our sample application are available at the following link:

<https://github.com/gitopsbook/sample-app-deployment>

Please go ahead and create a repository fork⁸⁴ as the first step. Flux requires write permissions to the deployment repository to automatically update image tags in the manifests.

NOTE *Reset your fork* Have you already forked the deployment repository while working on previous chapters? Please make sure to revert changes for the best experience. The simplest way is to delete the previously forked repository and fork it again.

Use `fluxctl` to install and configure the Flux daemon:

```
$ kubectl create ns flux
$ export GHUSER="YOURUSER"
$ fluxctl install \
--git-user=${GHUSER} \
--git-email=${GHUSER}@users.noreply.github.com \
--git-url=git@github.com:${GHUSER}/sample-app-deployment.git \
--git-path=. \
--namespace=flux | kubectl apply -f -
```

This command creates Flux daemon and configures it to deploy manifests from your Git repository. Make sure Flux daemon is running using the following command:

```
$ kubectl rollout status deploy flux -n flux
```

As part of this tutorial, we are going to try an automated repository update feature, so we need to give Flux repository write access. The convenient and secure way to provide write access to the Github repository is to use the deploy key.

⁸⁴ <https://help.github.com/en/github/getting-started-with-github/fork-a-repo>

NOTE Deploy Key A deploy key is an SSH key that is stored on your server and grants access to a single GitHub repository.

There is no need to generate the new SSH key manually. Flux generates a key during the first start and uses it to access the deployment repository. Run the following command to get generated SSH key:

```
$ fluxctl identity --k8s-fwd-ns flux
```

Navigate to <https://github.com/<username>/sample-app-deployment/settings/keys/new> and use the output of `fluxctl identity` command to create a new deployment key. Make sure to check the "Allow write access" checkbox to provide write access to the repository.

The configuration is done! While you are reading this, Flux should be cloning the repository and deploying manifests. Please go ahead and check the Flux daemon logs to confirm that.

Can you see `kubectl apply` in the logs?

```
$ kubectl logs deploy/flux -n flux -f
caller=sync.go:73 component=daemon info="trying to sync git changes to the cluster"
    old=6df71c4af912e2fc6f5fec5d911ac6ad0cd4529a
    new=1c51492fb70d9bdd2381ff2f4fdc51240dfe118
caller=sync.go:539 method=Sync cmd=apply args= count=2
caller=sync.go:605 method=Sync cmd="kubectl apply -f -" took=1.224619981s err=null
    output="service/sample-app configured\ndeleteDeployment.apps/sample-app configured"
caller=daemon.go:701 component=daemon event="Sync: 1c51492, default:service/sample-app"
    logupstream=false
```

Great, that means Flux successfully performed the deployment. Next, run to the following command to confirm that the sample app deployment resource has been created:

```
$ kubectl get deploy sample-app -n default
```

Congratulations! You successfully deployed your first application using Flux.

11.2.2 Observe application state

Reading logs of the Flux daemon is not the only way to get information about resources managed by Flux. The `fluxctl` CLI provides a set of commands that allow us to get detailed information about cluster resources. The first one we should try is `fluxctl list-workloads`. The command prints information about all Kubernetes resources that manage Pods in the cluster. Run the following command to output information about the `sample-app` deployment.

```
$ fluxctl list-workloads --k8s-fwd-ns flux
WORKLOAD      CONTAINER      IMAGE                  RELEASE
deployment/sample-app  sample-app  gitopsbook/sample-app:v0.1  ready
```

As you can see from the output, Flux is managing one deployment that creates a `sample-app` container using the `v0.1` version of the `gitopsbook/sample-app` image.

In addition to the information about the current image, Flux has scanned the Docker registry and collected all available image tags. Run the following command to print the list of discovered image tags:

```
$ fluxctl list-images --k8s-fwd-ns flux -w default:deployment/sample-app
WORKLOAD          CONTAINER   IMAGE           CREATED
deployment/sample-app sample-app gitopsbook/sample-app
                           | v0.2       27 Jan 20 05:46 UTC
                           | -> v0.1     27 Jan 20 05:35 UTC
```

From the command output, we can see that Flux correctly discovered both available image versions. In addition, Flux identified v0.2 as a newer version and ready to upgrade our deployment if we configure automated upgrades. Let's go ahead and do so.

11.2.3 Upgrade deployment image

By default, Flux does not upgrade the resource image version unless the resource has the `fluxcd.io/automated: 'true'` annotation. This annotation tells Flux that the resource image is managed automatically, and the image should be upgraded as soon as a new version is pushed to the Docker registry. The listing below contains the `sample-app` Deployment manifest with the applied annotation:

Listing 11.1 deployment.yaml.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
  annotations:
    fluxcd.io/automated: 'true'                                #A
spec:
  replicas: 1
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - command:
          - /app/sample-app
        image: gitopsbook/sample-app:v0.1                         #B
        name: sample-app
        ports:
          - containerPort: 8080
```

#A Annotation that enables automated management

#B The deployment image tag

One way to add the annotation is to manually edit the `deployment.yaml` file and commit it to the deployment repository. During the next reconciliation cycle, Flux should detect the

annotation and enable automated management. The `fluxctl` provides convenience commands, `automate` and `deautomate`, which can add or remove the annotation for you. Run the following command to automate the `sample-app` deployment management:

```
$ fluxctl automate --k8s-fwd-ns flux -w default:deployment/sample-app
WORKLOAD           STATUS   UPDATES
default:deployment/sample-app  success
Commit pushed: <commit-sha>
```

The command updates manifest and push the change into the Git repository. If you check the repository history using GitHub, you will see two commits where first commit updates the deployment annotation and second updates the image version.

Finally, let's verify the deployment status using the `fluxctl list-workflow` command:

```
$ fluxctl list-workloads --k8s-fwd-ns flux
WORKLOAD          CONTAINER    IMAGE                  RELEASE
deployment/sample-app  sample-app  gitopsbook/sample-app:v0.2  ready
```

The deployment image has been successfully updated to use the `v0.2` version of the `gitopsbook/sample-app` image. Don't forget to pull the changes executed by Flux into the local Git repository:

```
$ git pull
```

11.2.4 Use Kustomize for manifest generation

Managing plain YAML files in the deployment repository is not a very difficult task but not very practical in real life. As we've learned in previous chapters, it is common practice to maintain the base set of manifests for the application and generate environment specified manifests using tools like Kustomize or Helm. The integration with config management tools solves that problem, and Flux enables that feature using generators. Let's learn what generators are and how to use them.

Instead of providing first-class support for the selected set of config management tools, Flux provides the ability to configure the manifest generation process and integrate with any config management tool. The generator is a command that invokes the config management tool inside the Flux daemon that produces the final YAML. Generators are configured in the file named `.flux.yaml` stored in the deployment manifest repository.

Let's deep dive into the feature and learn configuration details on a real example. First of all, we need to enable the manifest generation in our Flux deployment. This is done using the `--manifest-generation` CLI flag of the Flux daemon. Run the following command to inject the flag into the Flux deployment using the JSON patch:

```
kubectl patch deploy flux --type json -n flux -p \
'[{ "op": "add", "path": "/spec/template/spec/containers/0/args/-", "value": "--manifest-generation"}]'
```

NOTE *JSON Patch*⁸⁵ JSON Patch is a format for describing changes to a JSON document. The patch document is a sequential list of operations that are applied to JSON objects and allow changes such as adding, removing, and replacing.

As soon as Flux configuration is updated, it is time to introduce Kustomize into our deployment repository and start leveraging it. Add the `kustomization.yaml` file using the following code from the code listing below:

Listing 11.2 kustomization.yaml.

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

resources:                      #A
- deployment.yaml
- service.yaml

images:
- name: gitopsbook/sample-app    #B
  newTag: v0.1
```

#A List of manifests included resource manifests

#B Transformer that changes image tag

The next step is to configure a generator that uses Kustomize. Add the following `.flux.yaml` file to the sample-app-deployment repository:

Listing 11.3 flux.yaml.

```
version: 1
patchUpdated:
  generators:
    - command: kustomize build .          # A
      patchFile: flux-patch.yaml        # B
                                         # C
```

#A List of generators

#B Generator command that leverages kustomize to generate manifests

#C Name of the file that stores manifest modifications; required for automated image updates

The configuration is done. Please go ahead and push changes to the deployment repository:

```
$ git add -A
$ git commit -am "Introduce kustomize" && git push
```

Let's take a look at `.flux.yaml` one more time and learn what was configured in detail. The generators section configures Flux to use `kustomize` for manifest generation. You can run the exact same command locally to verify that `kustomize` is producing the expected YAML manifests.

But what is the `patchFile` property? This is an updater configuration. To demonstrate how it works lets trigger the Flux release using the following commands:

⁸⁵ <http://jsonpatch.com/>

```
$ kubectl patch deploy sample-app -p '[{ "op": "add", "path": "/spec/template/spec/containers/0/image", "value": "gitopsbook/sample-app:v0.1"}]' - -type json -n default

$ fluxctl sync --k8s-fwd-ns flux
```

We've downgraded the `sample-app` Deployment back to the `v0.1` version and asked Flux to fix it. The `sync` command starts the reconciliation loop which once completed should update the image tag and push changes back to the Git repository. Since manifests are now generated using Kustomize, Flux no longer knows which file to update. The `patchFile` property specifies the file path within the deployment repository where image tag updates must be stored. The file contains the JSON merge patch that is automatically applied to the generator output.

NOTE JSON Merge Patch JSON Merge Patch is a JSON document that describes changes to be made to a target JSON and contains the nodes of the target document which should be different after the patch is applied.

Generated merge patch includes managed resources image changes. During the synchronization, process Flux generates and pushes the file with a merge patch to the Git repository and applies it on the fly to the generated YAML manifests.

Don't forget to pull the changes executed by Flux into the local Git repository:

```
$ git pull
```

11.2.5 Secure deployment using GPG

Flux is a very practical tool and focused on real-life use-cases. The deployment changes verification is one of such cases. As we've learned in Chapter 6, commits in the deployment repository should be signed and validated using a GPG key to ensure the author identity of the commit, preventing unauthorized changes to be pushed to the cluster.

The typical approach is to incorporate the GPG validation into the continuous integration pipeline. Flux provides this integration out-of-the-box, which saves time and provides a more robust implementation. The best way to learn how that feature operates is to try it.

First of all, we need a valid GPG key that can be used to sign and verify Git commits. If you've completed the Chapter 6 tutorials, then you already have a GPG key and can sign commits. Otherwise, please use the steps described in Appendix C to create the GPC key. After configuring the GPG key, we need to make it available to Flux and enable the commit verification.

To verify the commit, Flux needs to have access to which GPG key we trust. The key can be configured using a `ConfigMap`. Use the following command to create the `ConfigMap` and store your public key in it:

```
$ kubectl create configmap flux-gpg-public-keys -n flux --from-literal=author.asc="$(gpg --export --armor <ID>)"
```

The next step is to update the Flux deployment to enable the commit verification. Update the username in the `flux-deployment-patch.yaml` file represented in the listing below:

Listing 11.4 flux-deployment-patch.yaml.

```

spec:
  template:
    spec:
      volumes:
        - name: gpg-public-keys
          configMap:
            name: flux-gpg-public-keys
            defaultMode: 0400
      containers:
        - name: flux
          volumeMounts:
            - name: gpg-public-keys
              mountPath: /root/gpg-public-keys
              readOnly: true
      args:
        - --memcached-service=
        - --ssh-keygen-dir=/var/fluxd/keygen
        - --git-url=git@github.com:<USERNAME>/sample-app-deployment.git
        - --git-branch=master
        - --git-path=.
        - --git-label=flux
        - --git-email=<USERNAME>@users.noreply.github.com
        - --manifest-generation=true
        - --listen-metrics=:3031
        - --git-gpg-key-import=/root/gpg-public-keys           #C
        - --git-verify-signatures                           #D
        - --git-verify-signatures-mode=first-parent         #E

```

#A Kubernetes volume that uses ConfigMap as a data source

#B Volume mount that stores ConfigMap keys in /root/gpg-public-keys directory

#C The --git-gpg-key-import arg specifies the location of trusted GPG keys

#D The --git-verify-signatures arg enables commit verification

#E The --git-verify-signatures-modes=first-parent arg allows having unsigned commits in repo history

Apply the Flux deployment modifications using the following command:

```
$ kubectl patch deploy flux -n flux -p "$(cat ./flux-deployment-patch.yaml)"
```

Commit verification is now enabled. To prove it is working, please try triggering the syncing using the `fluxctl sync` command:

```
$ fluxctl sync --k8s-fwd-ns flux
Synchronizing with ssh://git@github.com/<USERNAME>/sample-app-deployment.git
Failed to complete sync job
Error: verifying tag flux: no signature found, full output:
error: no signature found

Run 'fluxctl sync --help' for usage.
```

The command expectedly fails since the most recent commit in the deployment repository is not signed. Lets go ahead and fix it. First create an empty signed Git commit using the command below and sync again:

```
$ git commit --allow-empty -S -m "verified commit"
$ git push
```

The next step is to sign the sync tag maintained by flux:

```
$ git tag --force --local-user=<GPG-KEY_ID> -a -m "Sync pointer" flux HEAD
$ git push --tags --force
```

The repository is successfully synced. Finally using the `fluxctl sync` command to confirm that verification is configured correctly.

```
fluxctl sync --k8s-fwd-ns flux
Synchronizing with ssh://git@github.com/<USERNAME>/sample-app-deployment.git
Revision of master to apply is f20ac6e
Waiting for f20ac6e to be applied ...
Done.
```

11.3 Multi-tenancy with Flux

Flux is a powerful and flexible tool but does not have features that are built specifically for multi-tenancy. So the question is can we use it: in a large organization with multiple teams. The answer is definitely yes. The Flux embraces the “Git push all the things” philosophy and relies on GitOps to manage multiple Flux instances deployed in a multi-tenant cluster.

In a multi-tenant cluster, the cluster users have only limited namespace access and cannot create new namespaces or any other cluster-level resources. Each team owns the namespace resources and performs operations on them independently from each other. In this case, it does not make sense to force everyone to use a single Git repository and rely on the infrastructure team to review every configuration change. On the other hand, the infrastructure team is responsible for overall cluster health and needs tools to manage cluster services. The application teams can still rely on Flux to manage the application resource. The infrastructure team uses Flux to provision namespaces as well as multiple Flux instances configured with the proper namespace level access. The figure below demonstrates the idea:

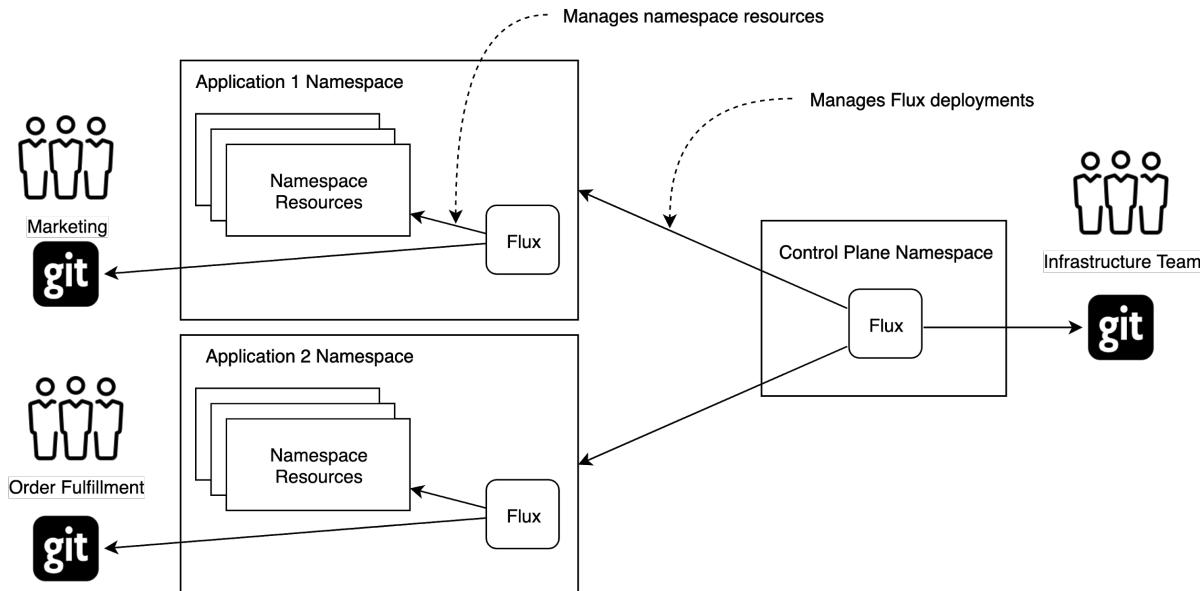
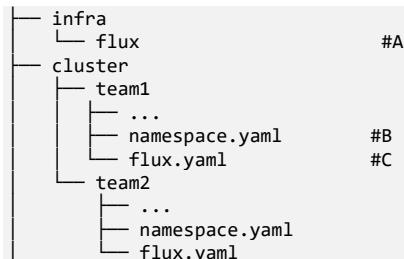


Figure 11.4 The cluster has one “control plane” namespace and a centralized cluster Git repository managed by the infrastructure team. The centralized repository contains manifests that represent centralized Flux deployment and team specific namespaces and Flux configurations.



#A Centralized Flux deployment that provision team specific namespaces
#B Team specific namespace manifest
#C Team specific Flux deployment

The application team can onboard themselves by creating the pull request to the centralized repository and adding the namespace and Flux manifests. As soon as pull request is merged the central Flux creates the namespace and provision team specific Flux ensuring the correct RBAC settings.

The team-specific Flux instance is configured to pull manifests from the separate Git repository that is managed by the application team. That means the application team is fully independent and doesn't have to involve the infrastructure team to update resources within their namespace.

11.4 Summary

- Flux is simple to install and maintain because Flux does not require new components and uses Kubernetes RBAC for access control.
- Flux can be configured with automated repository updates to automatically deploy new images.
- Because Flux interfaces directly with either Git or Docker Registry, Flux eliminates the need for custom integration in the CI pipeline for deployment.
- Flux comes with the CLI tool ‘fluxctl’ for Flux installation and deploying applications.
- Flux does not come with manifest generation tools but can easily integrate with tools such as Kustomize through simple configuration.
- Flux can easily integrate with GPG for secure deployment with simple configuration.
- Flux can be configured for multi-tenant through centralized provisioning of namespaces with access control and namespace specific Flux instances.

Appendix A

Setup a Test Kubernetes Cluster

A full production-capable Kubernetes cluster is a very complex system consisting of multiple components that must be installed and configured based on your particular needs. The topic about how to deploy and maintain Kubernetes in production goes way beyond the focus of this book and is covered elsewhere.

Luckily for us, there are several projects which handle the configuration complexity and allow running Kubernetes locally with a single CLI command. In this exercise you will learn how to deploy and run Kubernetes locally on your laptop. This is useful to get your hands dirty with Kubernetes and prepare you for completing the exercises in the remainder of this book. To the extent possible, all remaining exercises will utilize a cluster running on your laptop using an application called “minikube”. However, if you prefer to use your own cluster running on a cloud provider (or even on-premises) the exercises will work there as well.

NOTE `minikube` Minikube is an official tool maintained by the Kubernetes community to create a single-node Kubernetes cluster inside a VM on your laptop and supports macOS, Linux, and Windows. In addition to actually running the cluster, Minikube provides features which simplify accessing services inside Kubernetes, volume management and many more.

- **Minikube.**
- **Docker-for-Desktop.** If you are using docker on your laptop you might already have Kubernetes installed! Starting with version 18.6.0 both Windows and Mac Docker-for-Desktop comes with bundled Kubernetes binaries and developer productivity features.
- **K3S.** As the name implies, K3S is a lightweight Kubernetes deployment. According to authors K3S is five less than eight so K8S minus five is K3S. Besides funny name K3S is indeed extremely lightweight, fast and great choice if you need to run Kubernetes as part of CI job or on a hardware with limited resources. Installation instructions are available at <https://k3s.io>
- **KIND.** Another tool developed by Kubernetes community. KIND was developed by maintainers for Kubernetes v1.11+ conformance testing. Installation instructions are

available at <https://kind.sigs.k8s.io/>

While all listed tools simplify Kubernetes deployment to a single CLI command and provide great experience, Minikube is still the safest choice to get started with Kubernetes. With all platforms support, thanks to virtualization, and a great set of developer productivity features this is a great tool for both beginners and experts. All exercises and samples in this book rely on Minikube.

A.1 Prerequisites for working with Kubernetes

The following tools and utilities are needed to work with Kubernetes.

In addition to Kubernetes itself, we need to install kubectl. The kubectl is a command line utility which allows interacting with Kubernetes control plane and allows doing virtually anything with Kubernetes.

CONFIGURE KUBECTL

To get started go ahead and install Minikube as described at <https://kubernetes.io/docs/tasks/tools/install-kubectl/>. If you are macOS or Linux user you can complete the installation process in one step using homebrew package manager using the following command:

Listing A.1.

```
$ brew install kubectl
```

NOTE `homebrew` Homebrew is a free and open-source software package management system that simplifies the installation of software on Apple's macOS operating system and Linux. More information is available at <https://brew.sh/>

A.2 Install minikube and create a cluster

Minikube is an application that allows you to run a single-node Kubernetes cluster on your desktop or laptop machine. Installation instructions are available at <https://minikube.sigs.k8s.io/docs/start/>

<https://kubernetes.io/docs/tasks/tools/install-minikube/>

Most (but not all) exercises in this book can be completed using a local minikube cluster.

CONFIGURE MINIKUBE

The next step is to install and start the Minikube cluster. The installation process is described at <https://minikube.sigs.k8s.io/docs/start/>. The Minikube package is also available via Homebrew:

Listing A.2.

```
$ brew install minikube
```

If everything goes smooth so far then we are ready to start Minikube and configure our first deployment:

Listing A.3.

```
$ minikube start
  (minikube/default)
🕒 minikube v1.1.1 on darwin (amd64)
🔥 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
🐳 Configuring environment for Kubernetes v1.14.3 on Docker 18.09.6
🚜 Pulling images ...
🚀 Launching Kubernetes ...
⌛ Verifying: apiserver proxy etcd scheduler controller dns
🏁 Done! kubectl is now configured to use "minikube"
```

A.3 Create a GKE cluster in GCP

The Google Cloud Platform (GCP) offers the Google Kubernetes Engine (GKE) as part of their “Free Tier”.

<https://cloud.google.com/free/>

You can create a Kubernetes GKE cluster to run the exercises in this book.

<https://cloud.google.com/kubernetes-engine/>

Keep in mind that while GKE itself is free, you may be charged for other GCP resources that are created by Kubernetes. It is recommended that you delete your test cluster after completing each exercise in order to avoid unexpected costs.

A.4 Create an EKS cluster in AWS

Amazon Web Services (AWS) offers a managed Kubernetes service called Elastic Kubernetes Service (EKS). You can create a free AWS account and create an EKS cluster to run the exercises in this book. However, while relatively inexpensive, EKS is not a free service (it costs \$0.20/hour at the time of this writing) and you may also be charged for other resources created by Kubernetes. It is recommended that you delete your test cluster after completing each exercise in order to avoid unexpected costs.

There is a tool called eksctl by Weaveworks which allows you to easily create an EKS Kubernetes cluster in your AWS account.

<https://github.com/weaveworks/eksctl/blob/master/README.md>

Appendix B

Setup Gitops Tools

This appendix will go over the step by step instruction to set up tools required for the tutorials in Part III.

B.1 Install Argo CD

Argo CD supports several installation methods. You might use the official Kustomize based installation, the community maintained Helm chart⁸⁶, or even run the Argo CD operator⁸⁷ to manage the Argo CD deployments. The simplest possible installation method requires only to use a single YAML file. Please go ahead and use the following commands to install Argo CD into your minikube cluster.

```
$ kubectl create namespace argocd
$ kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
```

The command above installs all Argo CD components with the default settings that work for most users out of the box. For security reasons, the Argo CD UI and API are not exposed outside of the cluster by default. It is totally safe to open full access on minikube.

Enable load balancer access⁸⁸ in your minikube cluster by running the following command in a separate terminal:

```
$ minikube tunnel
```

Use the following command to open access to the “argocd-server” service and get the access URL:

```
$ kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'
```

⁸⁶ <https://github.com/argoproj/argo-helm/tree/master/charts/argo-cd>

⁸⁷ <https://github.com/argoproj-labs/argocd-operator>

⁸⁸ <https://minikube.sigs.k8s.io/docs/handbook/accessing/#loadbalancer-access>

Argo CD provides both a web-based user interface and a command-line interface (CLI). To simplify the instructions, we are going to use the CLI tool in this tutorial. Let's go ahead and install the CLI tool. You might use the following command to install Argo CD CLI on Mac or follow the official getting started instructions⁸⁹ to install the CLI on your platform:

```
$ brew tap argoproj/tap
$ brew install argoproj/tap/argocd
```

As soon as Argo CD is installed, it has a pre-configured admin user. The initial admin password is auto-generated to be the pod name of the Argo CD API server that can be retrieved using the command below:

```
$ kubectl get pods -n argocd -l app.kubernetes.io/name=argocd-server -o name | cut -d'/' -f 2
```

Use the following command to get the Argo CD server URL and update generated password using the Argo CD CLI:

```
$ argocd login <ARGOCD_SERVER-HOSTNAME>:<PORT>
$ argocd account update-password
```

The `<ARGOCD_SERVER-HOSTNAME>:<PORT>` is a minikube API and service port that should be obtained from the Argo CD URL. The URL might be retrieved using the following command:

```
minikube service argocd-server -n argocd --url
```

The command returns the HTTP service URL. Make sure to remove "http://" and use only hostname and the port to login using Argo CD CLI.

Finally, login to the Argo CD user interface. Please open the Argo CD URL in the browser and login using the admin username and your password. You are ready to go!

B.2 Install Jenkins X

Jenkins X CLI depends on kubectl⁹⁰ and Helm⁹¹ and will do its best to install those tools. However, the number of permutations of what we have on our laptops are close to infinite, so you're better off installing those tools first yourself.

NOTE At the time of this writing, February 2020, Jenkins X does not yet support Helm v3+. Please make sure that you're using Helm CLI v2+.

B.2.1 Prerequisites

You can use (almost) any Kubernetes cluster, but it needs to be publicly accessible. The main reason for that lies in GitHub triggers. Jenkins X relies heavily on GitOps principles. Most of the events will be triggered by GitHub webhooks. If your cluster cannot be accessed from

⁸⁹ https://argoproj.github.io/argo-cd/cli_installation/#download-with-curl

⁹⁰ <https://kubernetes.io/docs/tasks/tools/install-kubectl/>

⁹¹ https://docs.helm.sh/using_helm/#installing-helm

GitHub, you won't be able to trigger those events, and you will have difficulty following the examples.

Now, that poses two significant issues. You might prefer to practice locally using Minikube or Docker Desktop, but neither of the two is accessible from outside your laptop. You might have a corporate cluster that is inaccessible from the outside world. In those cases, I suggest you use a service from AWS, GCP, or from anywhere else. Finally, we'll perform some GitHub operations using the command `hub`. Install it if you don't have it already.

NOTE Please refer to Appendix A for more information on configuring AWS or GCP Kubernetes Cluster.

For your convenience, the list of all the tools we'll use is as follows.

- Git
- `kubectl`
- Helm⁹²
- AWS CLI
- `eksctl`⁹³ (if using AWS EKS)
- `gcloud` (if using Google GKE)
- `hub`⁹⁴

Now, let's install Jenkins X CLI.

```
$ brew tap jenkins-x/jx
$ brew install jx
```

B.2.2 Installing Jenkins X In Kubernetes Cluster

How can we install Jenkins X in a better way than what we're used to installing software? Jenkins X configuration should be defined as code and reside in a Git repository, and that's what the community created for us. It maintains a GitHub repository that contains the structure of the definition of the Jenkins X platform, together with a pipeline that will install it, as well as a requirements file that we can use to tweak it to our specific needs.

NOTE You can also refer to the Jenkins X site⁹⁵ on setting up Jenkins X in your Kubernetes cluster.

Let's take a look at the repository.

```
$ open "https://github.com/jenkins-x/jenkins-x-boot-config.git"
```

Once you see the repo in your browser, you will first create a fork under your GitHub account. We'll explore the files in the repo a bit later.

Next, we'll define a variable `CLUSTER_NAME` that will, as you can guess, hold the name of the cluster we created a short while ago. In the commands that follow, please replace the first occurrence of [...] with the name of the cluster and the second with your GitHub user.

⁹² https://docs.helm.sh/using_helm/#installing-helm

⁹³ <https://github.com/weaveworks/eksctl>

⁹⁴ <https://hub.github.com/>

⁹⁵ <https://jenkins-x.io/docs/getting-started/setup/>

```
$ export CLUSTER_NAME=[...]
$ export GH_USER=[...]
```

After we forked the Boot repo and we know how our cluster is called, we can clone the repository with a proper name that will reflect the naming scheme of our soon-to-be-installed Jenkins X.

```
$ git clone \
  https://github.com/$GH_USER/jenkins-x-boot-config.git \
  environment-$CLUSTER_NAME-dev
```

The key file that contains (almost) all the parameters that can be used to customize the setup is `jx-requirements.yml`. Let's take a look at it.

```
$ cd environment-$CLUSTER_NAME-dev
$ cat jx-requirements.yml
cluster:
  clusterName: ""
  environmentGitOwner: ""
  project: ""
  provider: gke
  zone: ""
gitops: true
environments:
- key: dev
- key: staging
- key: production
ingress:
  domain: ""
  externalDNS: false
  tls:
    email: ""
    enabled: false
    production: false
kaniko: true
secretStorage: local
storage:
  logs:
    enabled: false
    url: ""
  reports:
    enabled: false
    url: ""
  repository:
    enabled: false
    url: ""
versionStream:
  ref: "master"
  url: https://github.com/jenkins-x/jenkins-x-versions.git
webhook: prow
```

As you can see, that file contains values in a format that resembles the `requirements.yaml` file used with Helm charts. It is split into a few sections.

First, there is a group of values that define our cluster. You should be able to figure out what it represents by looking at the variables inside it. It probably won't take you more than

a few moments to see that we have to change at least some of those values, so that's what we'll do next.

Please open `jx-requirements.yml` in your favorite editor and change the following values.

Set `cluster.clusterName` to the name of your cluster. It should be the same as the name of the environment variable `CLUSTER_NAME`. If you already forgot it, execute `echo $CLUSTER_NAME`.

Set `cluster.environmentGitOwner` to your GitHub user. It should be the same as the one we previously declared as the environment variable `$GH_USER`.

Set `cluster.project` to the name of your GKE project, only if that's where your Kubernetes cluster is running. Otherwise, leave that value intact (empty).

Set `cluster.provider` to `gke` or to `eks` or to any other provider if you decided that you are brave and want to try currently unsupported platforms. Or, things may have changed since the writing of this chapter, and your provider is indeed supported now.

Set `cluster.zone` to whichever zone your cluster is running in. If you're running a regional cluster (as you should), then the value should be the region, not the zone. If, for example, you used my Gist to create a GKE cluster, the value should be `us-east1-b`. Similarly, the one for EKS is `us-east-1`.

We're finished with the `cluster` section, and the next in line is the `gitops` value. It instructs the system how to treat the Boot process. I don't believe it makes sense to change it to `false`, so we'll leave it as-is (true).

The next section contains the list of the environments that we're already familiar with. The keys are the suffixes, and the final names will be a combination of `environment-` with the name of the cluster followed by the key. We'll leave them intact.

The `ingress` section defines the parameters related to external access to the cluster (domain, TLS, etc.).

The `kaniko` value should be self-explanatory. When set to `true`, the system will build container images using Kaniko instead of, let's say, Docker. That is a much better choice since Docker cannot run in a container and, as such, poses a significant security risk (mounted sockets are evil), and it messes with Kubernetes scheduler given that it bypasses its API. In any case, Kaniko is the only supported way to build container images when using Tekton, so we'll leave it as-is (true).

Next, we have `secretStorage` currently set to `local`. The whole platform will be defined in this repository, except for secrets (e.g., passwords). Pushing them to Git would be childish, so Jenkins X can store the secrets in different locations. If we'd change it to `local`, that location is your laptop. While that is better than a Git repository, you can probably imagine why that is not the right solution. Keeping secrets locally complicates cooperation (they exist only on your laptop), is volatile, and is only slightly more secure than Git. A much better place for secrets is HashiCorp Vault. It is the most commonly used solution for secrets management in Kubernetes (and beyond), and Jenkins X supports it out of the box. If you have a vault setup, you can set the value of `secretStorage` to `vault`. Otherwise, you can leave the default value '`local`'.

Below the `secretStorage` value is the whole section that defines storage for logs, reports, and repositories. If enabled, those artifacts will be stored on a network drive. As you already know, containers and nodes are short-lived, and if we want to preserve any of those, we need to store them elsewhere. That does not necessarily mean that network drives are the best place, but rather that's what comes out of the box. Later on, you might choose to change that and, let's say, ship logs to a central database like ElasticSearch, PaperTrail, CloudWatch, StackDriver, etc.

For now, we'll keep it simple and enable network storage for all three types of artifacts.

Set the value of `storage.logs.enabled` to `true`.

Set the value of `storage.reports.enabled` to `true`.

Set the value of `storage.repository.enabled` to `true`.

For now, we'll keep it simple and keep the default values (`true`) that enable network storage for all three types of artifacts.

The `versionsStream` section defines the repository that contains versions of all the packages (charts) used by Jenkins X. You might choose to fork that repository and control versions yourself. Before you jump into doing just that, please note that Jenkins X versioning is quite complex, given that many packages are involved. Leave it be unless you have a very good reason to take over control of the Jenkins X versioning and that you're ready to maintain it.

As you already know, Prow only supports GitHub. If that's not your Git provider, Prow is a no-go. As an alternative, we could set it up in Jenkins, but that's not the right solution either. Jenkins (without X) is not going to be supported for long, given that the future is in Tekton. It was used in the first generation of Jenkins X only because it was a good starting point and because it supports almost anything we can imagine. But, the community has embraced Tekton as the only pipeline engine, and that means that static Jenkins X is fading away and that it is used mostly as a transition solution for those accustomed to the "traditional" Jenkins.

So, what can we do if Prow is not a choice if you do not use GitHub, and Jenkins days are numbered? To make things more complicated, even Prow will be deprecated sometime in the future (or past depending when you read this). It will be replaced with Lighthouse, which, at least at the beginning, will provide similar functionality as Prow. Its primary advantage when compared with Prow is that Lighthouse will (or already does) support all major Git providers (e.g., GitHub, GitHub Enterprise, Bitbucket Server, Bitbucket Cloud, GitLab, etc.). At some moment, the default value of `webhook` will be `lighthouse`. But, at the time of this writing (October 2019), that's not the case since Lighthouse is not yet stable and production-ready. It will be soon. Or, maybe it already is, and I did not yet rewrite this chapter to reflect that.

In any case, we'll keep Prow as our webhook (for now).

Please only execute the following commands if you are using EKS. They will add additional information related to Vault, namely the IAM user that has sufficient permissions to interact with it. Make sure to replace [...] with your IAM user that has sufficient permissions (being admin always works).

```
$ export IAM_USER=[...] # e.g., jx-boot
echo "vault:
aws:"
```

```
autoCreate: true
iamUserName: \"${IAM_USER}\" \
| tee -a jx-requirements.yml
```

Please only execute the following commands if you are using EKS. The `jx-requirements.yaml` file contains zone entry, and for AWS we need a region. That command will replace one with the other.

```
$ cat jx-requirements.yml \
| sed -e \
's@zone@region@g' \
| tee jx-requirements.yml
```

Let's take a peek at how `jx-requirements.yaml` looks like now.

```
$ cat jx-requirements.yaml
cluster:
  clusterName: "jx-boot"
  environmentGitOwner: "vfarcic"
  project: "devops-26"
  provider: gke
  zone: "us-east1"
gitops: true
environments:
- key: dev
- key: staging
- key: production
ingress:
  domain: ""
  externalDNS: false
  tls:
    email: ""
    enabled: false
    production: false
kaniko: true
secretStorage: vault
storage:
  logs:
    enabled: true
    url: ""
  reports:
    enabled: true
    url: ""
  repository:
    enabled: true
    url: ""
versionStream:
  ref: "master"
  url: https://github.com/jenkins-x/jenkins-x-versions.git
webhook: prow
```

Now, you might be worried that we missed some of the values. For example, we did not specify a domain. Does that mean that our cluster will not be accessible from outside? We also did not specify the URL for storage. Will Jenkins X ignore it in that case?

The truth is that we specified only the things we know. For example, if you created a cluster using my Gist, there is no Ingress, so there is no external load balancer that it was

supposed to create. As a result, we do not yet know the IP through which we can access the cluster, and we cannot generate a .nip.io domain. Similarly, we did not create storage. If we did, we could have entered addresses into URL fields.

Those are only a few examples of the unknowns. We specified what we know, and we'll let Jenkins X Boot figure out the unknowns. Or, to be more precise, we'll let Boot create the resources that are missing and thus convert the unknowns into known.

Let's install Jenkins X.

```
$ jx boot
```

Now we need to answer quite a few questions. In the past, we tried to avoid answering questions by specifying all answers as arguments to commands we were executing. That way, we had a documented method for doing things that do not end up in a Git repository. Someone else could reproduce what we did by running the same commands. This time, however, there is no need to avoid questions since everything we'll do will be stored in a Git repository.

The first input is asking for a comma-separated list of Git provider usernames of approvers for the development environment repository. That will create the list of users who can approve pull requests to the development repository managed by Jenkins X Boot. For now, type your GitHub user and hit the enter key.

We can see that, after a while, we were presented with two warnings stating that TLS is not enabled for Vault and webhooks. If we specified a "real" domain, Boot would install Let's Encrypt and generate certificates. But, since I couldn't be sure that you have a domain at hand, we did not specify it, and, as a result, we will not get certificates. While that would be unacceptable in production, it is quite OK as an exercise.

As a result of those warnings, the Boot is asking us whether we wish to continue. Type `y` and press the enter key to continue.

Given that Jenkins X creates multiple releases a day, the chances are that you do not have the latest version of `jx`. If that's the case, the Boot will ask, would you like to upgrade to the `jx` version?. Press the enter key to use the default answer `Y`. As a result, the Boot will upgrade the CLI, but that will abort the pipeline. That's OK. No harm done. All we have to do is repeat the process but, this time, with the latest version of `jx`.

```
$ jx boot
```

The process started again. We'll skip commenting on the first few questions from `jx boot` and continue without TLS. Answers are the same as before (`y` in both cases).

The next set of questions is related to long term storage for logs, reports, and repository. Press the enter key to all three questions, and the Boot will create buckets with auto-generated unique names.

From now on, the process will create the secrets and install CRDs (Custom Resource Definitions) that provide custom resources specific to Jenkins X. Then, it'll install nginx Ingress (unless your cluster already has one) and set the domain to .nip.io since we did not specify one. Further on, it will install CertManager, which will provide Let's Encrypt certificates. Or, to be more precise, it would provide the certificates if we specified a domain.

Nevertheless, it's installed just in case we change our minds and choose to update the platform by changing the domain and enabling TLS later on.

The next in line is Vault. The Boot will install it and attempt to populate it with the secrets. But, since it does not know them just yet, the process will ask us another round of questions. The first one in this group is the Admin Username. Feel free to press the enter key to accept the default value admin. After that comes Admin Password. Type whatever you'd like to use (we won't need it today).

The process will need to know how to access our GitHub repositories, so it asks us for the Git username, email address, and token. You can use your github username and email for the first two questions. As for the token⁹⁶, you'll need to create a new one in GitHub and **grant full repo access**. Finally, the next question related to secrets is HMAC token. Feel free to press the enter key, and the process will create it for you.

Finally comes the last question. Do you want to configure an external Docker Registry? Press the enter key to use the default answer (N) and the Boot will create it inside the cluster or, as in case of most cloud providers, use the registry provided as a service. In the case of GKE, that would be GCR, for EKS that's ECR. In any case, by not configuring an external Docker Registry, the Boot will use whatever makes the most sense for a given provider.

```
? Jenkins X Admin Username admin
? Jenkins X Admin Password [? for help] *****
? The Git user that will perform git operations inside a pipeline. It should be a user
    within the Git organisation/own? Pipeline bot Git username vfarcic
? Pipeline bot Git email address vfarcic@gmail.com
? A token for the Git user that will perform git operations inside a pipeline. This
    includes environment repository creation, and so this token should have full
    repository permissions. To create a token go to
    https://github.com/settings/tokens/new?scopes=repo,read:user,read:org,user:email,wri
    te:repo_hook,delete_repo then enter a name, click Generate token, and copy and paste
    the token into this prompt.
? Pipeline bot Git token *****
Generated token bb65edc3f137e598c55a17f90bac549b80fefbcraf, to use it press enter.
This is the only time you will be shown it so remember to save it
? HMAC token, used to validate incoming webhooks. Press enter to use the generated token [?
    for help]
? Do you want to configure non default Docker Registry? No
```

The rest of the process will install and configure all the components of the platform. We won't go into all of them since they are the same as those we used before. What matters is that the system will be fully operational a while later.

The last step will verify the installation. You might see a few warnings during this last step of the process. Don't be alarmed. The Boot is most likely impatient. Over time, you'll see the number of running Pods increasing and those that are pending decreasing, until all the Pods are running.

That's it. Jenkins X is now up-and-running. We have the whole definition of the platform with complete configuration (except for secrets) stored in a Git repository.

```
verifying the Jenkins X installation in namespace jx
verifying pods
Checking pod statuses
```

⁹⁶ <https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token>

```

POD                                         STATUS
jenkins-x-chartmuseum-774f8b95b-bdxfh      Running
jenkins-x-controllerbuild-66cbf7b74-twkbp    Running
jenkins-x-controllerrole-7d76b8f449-5f5xx    Running
jenkins-x-gcactivities-1594872000-w6gns      Succeeded
jenkins-x-gcpods-1594872000-m7kgq          Succeeded
jenkins-x-heapster-679ff46bf4-94w5f         Running
jenkins-x-nexus-555999cf9c-s8hnn           Running
lighthouse-foghorn-599b6c9c87-bvpct        Running
lighthouse-gc-jobs-1594872000-wllsp       Succeeded
lighthouse-keeper-7c47467555-c87bz        Running
lighthouse-webhooks-679cc6bbbb-fxw7z       Running
lighthouse-webhooks-679cc6bbbb-z14bw       Running
tekton-pipelines-controller-5c4d79bb75-75hvj Running

Verifying the git config
Verifying username billyy at git server github at https://github.com
Found 2 organisations in git server https://github.com: IntuitDeveloper, intuit
Validated pipeline user billyy on git server https://github.com
Git tokens seem to be setup correctly
Installation is currently looking: GOOD
Using namespace 'jx' from context named 'gke_hazel-charter-283301_us-east1-b_cluster-1' on
server 'https://34.73.66.41'.

```

B.3 Install Flux

Flux consists of a CLI client and daemon that runs inside of the managed Kubernetes cluster. This document explains how to install the Flux CLI only. The daemon installation requires to specify the Git repository with access credentials and covered in the Chapter 11.

B.3.1 Install CLI client

The Flux distribution includes the CLI client named fluxctl. The fluxctl automates Flux daemon installation and allows to get the information about Kubernetes resources controlled by the Flux daemon.

Use one of following commands to install the fluxctl on Mac, Linux and Windows:

Mac OS

```
brew install fluxctl
```

Linux

```
sudo snap install fluxctl
```

Windows

```
choco install fluxctl
```

Find more information about fluxctl installation details in the official installation instructions: <https://docs.fluxcd.io/en/latest/references/fluxctl/>

Appendix C

Configure GPG Key

GPG, or GNU Privacy Guard, is a public key cryptography implementation. GPG allows for the secure transmission of information between parties and can be used to verify that the origin of a message is genuine.

C.1 Configure GPG Key

1. First of all, we need to install the GPG command-line tool. Regardless of your operating system, the process might take a while. Mac OS users might use the brew package manager to install GPC with the following command:

```
brew install gpg
```

2. The next step is a GPG key generation that will be used to sign and verify commits. Use the command below to generate a key. At the prompt press Enter to accept default key settings. While entering the user identity information, make sure to use the verified email of your GitHub account.

```
gpg --full-generate-key
```

3. Find the ID of the generated key and use the ID to access the GPG key body.

```
gpg --list-secret-keys --keyid-format LONG
gpg --armor --export <ID>
```

In this example, the GPG key ID is 3AA5C34371567BD2:

```
gpg --list-secret-keys --keyid-format LONG
/Users/hubot/.gnupg/secreting.gpg
-----
sec 4096R/3AA5C34371567BD2 2016-03-10 [expires: 2017-03-10]
uid                               Hubot
ssb 4096R/42B317FD4BA89E7A 2016-03-10
```

4. Use the output of `gpg --export` and follow the steps described at <https://help.github.com/en/github/authenticating-to-github/adding-a-new-gpg-key-to-your-github-account> to add the key to your Github account.
5. Configure git to use generated GPG key:

```
git config --global user.signingkey <ID>
```